An intelligent framework to explore and detect community smells in software engineering

by

Nuri ALMARIMI

MANUSCRIPT-BASED THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, NOVEMBER 22, 2023

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

Nuri Almarimi, 2023

This Creative Commons license allows readers to download this work and share it with others as long as the author is credited. The content of this work cannot be modified in any way or used commercially.

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Ali Ouni, Thesis supervisor
Department of Software and IT Engineering, École de technologie supérieure

Mr. Mohammad Jahazi, Chair, Board of Examiners
Department of Mechanical Engineering, École de technologie supérieure

Mr. Segla Kpodjedo, Member of the Jury
Department of Software and IT Engineering, École de technologie supérieure

Mr. Eray Tüzün, External Examiner
Department of Computer Engineering, Bilkent University

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON " NOVEMBER 01, 2023"

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to my supervisor, Ali Ouni, for his guidance, expertise, and unwavering support throughout this research journey. His invaluable insights, constructive feedback, and dedication have been instrumental in shaping the direction and quality of this thesis. Without him, the achievement of this work would not have been possible.

I would like to take this opportunity to express my gratitude to all those who have provided financial support for my studies and my stay in Canada: the Libyan Higher Education Ministry and the College of Electronic Technology / Bani Walid.

I would also like to extend my gratitude to all my co-authors who have contributed to the completion of this thesis.

I am deeply grateful to all the members of the jury for agreeing to read the manuscript and participating in the defense of this thesis.

Overall, I want to express my warmest thanks to my mother, my wife, my brothers, and my child, Yousuf, for their unconditional love and support in every possible way. My most sincere thanks to all my friends, both back in Libya and the ones I have made here in Canada, who have always supported me in various ways and shared many fun and stress-relieving moments, as well as their encouragement and support.

Un cadre intelligent pour explorer et détecter les odeurs de la communauté en génie logiciel.

Nuri ALMARIMI

RÉSUMÉ

L'ingénierie logicielle peut être définie comme l'effort coordonné de diverses entités, y compris des organisations et des individus, pour construire des produits logiciels. Ainsi, la structure sociale au sein d'une communauté de développement logiciel, y compris les interactions entre les développeurs, est cruciale pour le succès des projets logiciels. Les communautés de projets sont de plus en plus sollicitées dans les systèmes logiciels modernes pour améliorer la productivité des développeurs et garantir la livraison de logiciels de haute qualité. Cependant, ces communautés manquent souvent d'un suivi suffisant de leurs structures organisationnelles, en particulier dans les communautés plus grandes avec des structures explicites où le soutien managérial peut faire défaut. Les communautés de développement logiciel doivent donc coordonner activement leurs efforts pour favoriser une collaboration efficace et promouvoir le bien-être de la communauté de développement. Des recherches récentes ont introduit le terme "odeurs de la communauté" pour décrire une série de modèles socio-techniques ayant un effet préjudiciable sur la santé organisationnelle d'un projet. Les odeurs de la communauté sont associées à des circonstances résultant de mauvaises pratiques organisationnelles et sociales, ce qui contribue à l'accumulation de dettes sociales. Cette thèse présente une série d'études empiriques visant à comprendre les défis de la détection des odeurs de la communauté dans le domaine de l'ingénierie logicielle, à mettre en évidence la valeur des odeurs de la communauté dans le développement logiciel et à proposer de nouvelles approches pour soutenir les développeurs et les gestionnaires dans l'amélioration de l'efficacité de l'ingénierie logicielle au sein de communautés sub-optimales.

Plus précisément, nous abordons trois aspects du problème de détection des odeurs de la communauté. Tout d'abord, nous présentons une étude pour explorer et détecter les odeurs de la communauté dans des projets open-source. Grâce à cette étude, nous développons un modèle utilisant une approche d'apprentissage automatique qui apprend à partir d'un ensemble de symptômes organisationnels-sociaux pour caractériser la présence potentielle d'odeurs de la communauté. Ensuite, nous menons une étude empirique qui applique un modèle d'apprentissage multi-étiquettes (MLL) pour traiter les symptômes entrelacés des odeurs de la communauté existantes. Enfin, nous proposons un cadre et une approche pour améliorer la détection des odeurs de la communauté dans le domaine de l'ingénierie logicielle. Notre cadre intègre plusieurs sources de données, notamment l'analyse des réseaux sociaux, l'analyse des sentiments et les métriques de facteur de camion. Enfin, nous introduisons un nouvel outil automatisé basé sur le cadre proposé pour détecter les odeurs de la communauté dans des projets open-source. Cette approche surpasse les techniques existantes et les approches de pointe.

Les résultats de la thèse mettent en évidence un changement significatif dans le paysage de la communauté de développement logiciel. Les résultats présentés fournissent de nouvelles perspectives sur la détection des odeurs de la communauté au sein de la communauté de

développement logiciel. De plus, ils mettent en lumière des approches novatrices pour la détection précoce des odeurs de la communauté potentielles dans un projet logiciel, qui peuvent être utilisées par la communauté logicielle, y compris les développeurs, les gestionnaires et les organisations.

Mots-clés: Community smells, Dette sociale, Génie logiciel basé sur la recherche, Génie logiciel empirique, Apprentissage

An intelligent framework to explore and detect community smells in software engineering

Nuri ALMARIMI

ABSTRACT

Software engineering can be defined as the coordinated effort of various entities, including organizations, and individuals, to build software products. Thus, the social structure within a software development community, including the interactions among developers, is crucial for the success of software projects. Project communities are increasingly relied upon in modern software systems to enhance developers' productivity and ensure the delivery of high-quality software. However, such communities often lack sufficient monitoring of their organizational structures, especially in larger communities with explicit structures where managerial support may be lacking. Software development communities, therefore, need to actively coordinate their efforts to foster effective collaboration and promote the well-being of the development community. Recent research has introduced the term 'community smells' to describe a range of socio-technical patterns that have a detrimental effect on the organizational health of a project. Community smells are associated with circumstances resulting from poor organizational and social practices, which contribute to the accumulation of social debt. This thesis presents a series of empirical studies that aim to understand the challenges of detecting community smells in the software engineering domain, highlight the value of community smells in software development, and propose novel approaches to support developers and managers in improving the efficiency of software engineering within sub-optimal communities.

Specifically, we address three aspects of the community smells detection problem. First, we present a study to explore and detect community smells in open-source projects. Through this study, we develop a model using a machine-learning approach that learns from a set of organizational-social symptoms to characterize the potential presence of community smells. Second, we conduct an empirical study that applies a multi-label learning (MLL) model to deal with the interleaving symptoms of existing community smells. Third, we propose a framework and approach to enhance the detection of community smells in the software engineering domain. Our framework integrates multiple data sources, including social network analysis, sentiment analysis, and truck factor metrics. Finally, we introduce a novel automated tool based on the proposed framework for detecting community smells in open-source projects. This approach outperforms existing techniques and state-of-the-art approaches.

The findings of the thesis highlight a significant shift in the software development community landscape. The presented results provide fresh insights into the detection of community smells within the software development community. Furthermore, they shed light on innovative approaches for the early detection of potential community smells in a software project, which can be utilized by the software community, including developers, managers, and organizations.

Keywords: Community smells, Social debt, Search-based Software Engineering, Empirical Software Engineering, Machine learning.

TABLE OF CONTENTS

Page

INTRODUCTION 1
0.1 Research context 1
0.2 Problem statement 2
0.2.1 Problem 1: Automating community smells detection. 3
0.2.2 Problem 2: Limited empirical knowledge of interleaving symptoms of existing community smells types. 3
0.2.3 Problem 3: Lack of empirical knowledge of developers' communications. 4
0.3 Research objectives 4
0.4 Main contributions 5
0.5 Thesis organization 6
0.6 Publications 7

CHAPTER 1 STATE OF THE ART 9
1.1 Introduction 9
1.2 Background 9
1.2.1 Social debt and community smells 9
1.2.2 Developer social networks 10
1.2.3 Sentiment Analysis for Software Engineering 12
1.2.4 Search-based software engineering (SBSE) 14
1.3 Related Work 14
1.3.1 Open source community's health 15
1.3.2 Community smells detection approaches and tools 16
1.3.3 Sentiment analysis detection approaches and tools 18
1.4 Chapter Summary 20

CHAPTER 2 LEARNING TO DETECT COMMUNITY SMELLS IN OPEN SOURCE SOFTWARE PROJECTS 21
2.1 Introduction 22
2.2 Background 25
2.2.1 Community smells definitions 25
2.3 csDetector: Community smells detection using machine learning 27
2.3.1 Data collection 27
2.3.2 Metrics framework 29
2.3.3 Training model 32
2.3.4 Data preparation pipeline 33
2.4 Empirical study setup 34
2.5 Results and discussions 40
2.6 Threats to validity 53
2.6.1 Construct validity 53

2.6.2 Internal validity 54
2.6.3 External validity 55
2.7 Chapter Summary 56

CHAPTER 3 ON THE DETECTION OF COMMUNITY SMELLS USING GENETIC PROGRAMMING-BASED ENSEMBLE CLASSIFIER CHAIN 57
3.1 Introduction 58
3.2 Background 61
3.2.1 Community Smells Definitions 61
3.2.2 Search Based Software Engineering 63
3.2.3 Multi-label learning 63
3.3 Approach 65
3.3.1 Problem formulation 65
3.3.2 Approach Overview 65
3.3.3 Phase A : Training data collection 67
3.3.4 Phase B : Features Extraction Module 70
3.3.5 Phase C : Genetic Programming-based Ensemble Classifier Chain (GP-ECC) 73
3.3.6 Phase D : Detection Phase 77
3.4 Evaluation 77
3.4.1 Research Questions 77
3.4.2 Analysis method 77
3.4.3 Results 80
3.5 Threats to validity 82
3.6 Chapter Summary 83

CHAPTER 4 IMPROVING THE DETECTION OF COMMUNITY SMELLS THROUGH SOCIO-TECHNICAL AND SENTIMENT ANALYSIS 85
4.1 Introduction 86
4.1.1 Novelty statement 89
4.1.2 Replication package 90
4.1.3 Paper organization 90
4.2 Background 90
4.2.1 Community Smells in Open Source Software 90
4.2.2 Search Based Software Engineering 93
4.2.3 Multi-Label Learning 93
4.3 Approach 96
4.3.1 Step A : Training data collection 97
4.3.2 Step B: Features Extraction Module 100
4.3.3 Step C: Genetic Programming-based Ensemble Classifier Chain (GP-ECC) 105
4.3.4 Step D : Detection Phase 110

4.4 Empirical Evaluation 111
4.4.1 Research Questions 111
4.4.2 Analysis method 111
4.4.3 Results 114
4.5 Threats to validity 125
4.6 Chapter Summary 127

CHAPTER 5 DISCUSSION AND LESSONS LEARNED 129
5.1 Discussion 129
5.2 Lessons Learned 130

CONCLUSION AND RECOMMENDATIONS 133
6.1 Conclusion and Findings 133
6.2 Future Work 134

BIBLIOGRAPHY 136

LIST OF TABLES

Page

Table 2.1 Dataset statistics. 29

Table 2.2 Metrics framework. 30

Table 2.3 Algorithms parameters configuration. 36

Table 2.4 The confusion matrix to calculate the accuracy. 38

Table 2.5 The average algorithms performance in terms of Accuracy, Kappa statistic, RMSE, and AUC. 41

Table 2.6 The achieved results 42

Table 2.7 A summary of the identified smells in each studied system. 44

Table 2.8 Examples of projects and related smells. 45

Table 2.9 The comparison results of csDetector, CodeFace4smell and Truck Factor tools for community smells detection in terms of detection accuracy. 47

Table 2.10 The analysis results of the most influential metrics on community smells detection. 49

Table 2.11 The achieved accuracy, Kappa and RMSE results by csDetector with and without the newly considered metrics. 51

Table 3.1 Questionnaire filled in by the study participants. 68

Table 3.2 Socio-technical metrics framework. 72

Table 3.3 Dataset statistics. 78

Table 3.4 The achieved results by each of the meta-algorithms ECC, RAKEL and BR with their base learning algorithms GP, J48, and RF; and ML.KNN. 81

Table 3.5 The most influential features for each smell. 82

Table 4.1 Questionnaire filled in by the study participants. 98

Table 4.2 Socio-technical metrics framework. 104

Table 4.3 Dataset statistics. 112

Table 4.4 The achieved results by each of the meta-algorithms ECC, RAKEL and BR with their base learning algorithms GP, J48, and RF; and ML.KNN. 116

Table 4.5 Examples of projects and their related community smells. 118

Table 4.6 The most influential features for each community smell type. 124

LIST OF FIGURES

Page

Figure 0.1 Thesis organization 7

Figure 2.1 An overview of the proposed csDetector approach. 27

Figure 2.2 Beanplots for the distribution of smells, developers, commits, and lifetime among the studied projects. 29

Figure 2.3 An overview of our social-organizational metrics framework. 32

Figure 2.4 Visualization of the metrics correlation analysis. 33

Figure 2.5 Boxplots of the achieved Accuracy, AUC, Kappa, and RMSE results for the eight considered community smells by each of the considered machine learners, C4.5, JRip, LibSVM, Naïve Bayes, Random Forest and SMO. 42

Figure 2.6 Results of the influential instances analysis on the models accuracy. 52

Figure 3.1 The community smells detection framework using the GP-ECC model. 67

Figure 3.2 An overview of the features extraction module. 71

Figure 3.3 A simplified example of a solution for OSE smell. 75

Figure 3.4 The achieved precision and recall scores by GP-ECC for each communiy smell type. 82

Figure 4.1 A real-world example of SD smell. 96

Figure 4.2 The community smells detection framework using the GP-ECC model. 97

Figure 4.3 An overview of the features extraction module. 101

Figure 4.4 A simplified example of a solution for OSE smell. 107

Figure 4.5 A simplified example of a crossover operator. 109

Figure 4.6 A simplified example of a mutation operator. 110

Figure 4.7 The achieved precision and recall scores by eGP-ECC for each community smell type. 119
Figure 4.8 Sadness comment sentiments as an example. 120
Figure 4.9 Anger comment sentiments as an example. 121
Figure 4.10 Negative emotion comment as an example. 121
Figure 4.11 Brief comment as an example. 121
Figure 4.12 The achieved precision and recall results by eGP-ECC (with the new metrics) and GP-ECC (without the new metrics). 122

INTRODUCTION

0.1 Research context

Software engineering has increasingly come to depend on open-source communities more than ever before. However, the nature of such communities often involves organizational structures with limited monitoring (Tamburri, Palomba, Serebrenik & Zaidman (2019c)). For larger communities with an explicit structure, there is often insufficient managerial support.(Tamburri, Kazman & Fahimi (2016)). Hence, software development communities are required to coordinate their efforts to provide effective collaboration to produce the well-being of the development community (Tamburri, Kruchten, Lago & Vliet (2015b)). The social aspects related to the organizational structure of a software development community and a project social network are the most important factors that influence the start-up and continued success of the software project (Tamburri *et al.* (2019c)). Recent studies on open-source community failure demonstrate an increasing need for automated and semi-automated support to measure the social, organizational, and socio-technical characteristics of such communities (Tamburri, Kruchten, Lago & van Vliet (2013)).

Community researchers, such as architects guiding the social and organizational structure of development projects (Tamburri *et al.* (2016)), could utilize tools to identify undesirable changes and organizational distress within a community. This would enable them to make informed decisions aimed at reorganizing the community (Tamburri *et al.* (2019c)). In fact, by using tools such as automated monitoring systems that observe people and software artifacts together, community researchers could gain a better understanding of the characteristics of other open-source communities and then replicate these characteristics in their own communities.

Improving the quality of software projects success involves detecting community smells. Typically, community smells are unfavorable conditions in open-source communities that can

result in social debt, which is an unexpected project cost associated with suboptimal development communities (Tamburri, Kruchten, Lago & van Vliet (2013b)). Social debt can cause ripple effects leading to technical debt (Tamburri, Palomba & Kazman (2019b)). One of the widely used techniques to detect community smells is social network analysis. Metrics such as density, centrality, betweenness and closeness in a developer social network are relevant factors to consider when detecting and predicting the emergence of community smells. These metrics indicate that the community structure can impact the occurrence of smells.(Palomba & Tamburri (2021)).

0.2 Problem statement

Software products are the result of collaborative efforts by teams working together to produce high-quality software. Negative organizational efforts can lead to poor software quality. Hence, communication and coordination in large software development teams must be well organized within software development communities. Understanding the nature of these communities is crucial to ensuring the success of software projects. The socio-technical decisions related to the organizational structure of a software development community are the most important factors that affect how people work and interact with each other. Therefore, detecting community smells in negative community structures can have a significant impact on the success of software projects and can help improve the quality of software products and the software development process. These concerns and motivations led to the formulation of this thesis problem statement, which is stated as follows:

"With the nature of open-source communities and given that the community smells can have a negative impact on project success, we believe that there is untapped potential in detecting and exploring community smells within software development communities".

In this section, we highlight the different problems addressed in this thesis that are related to the social aspects connected to sub-optimal organizational communities as follows:

0.2.1 Problem 1: Automating community smells detection.

Like any software detection problem, there are many open issues and challenges that need to be addressed when defining a detection strategy. In the following, we highlight these open issues.

- Although there is a definition of community smells in an industrial environment, there is no consensus on the definition of community smells in open-source projects. If such smells definitions exist, translating these definitions into detection rules becomes a subject of community smell detection.
- Community smells detection requires an automatic approach to define detection rules for smells. The process of manually defining detection rules, and exploring the community-smell candidates, is time-consuming, tedious, and not always profitable. Furthermore, community smells detection requires an expert to manually write and validate detection rules.

0.2.2 Problem 2: Limited empirical knowledge of interleaving symptoms of existing community smells types.

- Detecting a set of symptoms for each community smell type does not necessarily mean that the actual community smell is effectively detected; moreover, in most cases, different community smells exhibit similar symptoms. The majority of existing detection methods do not consider the interleaving symptoms of existing community smell types. In these cases, the same symptom could be associated with multiple smell types, which compromises the precise identification of community smell types.

0.2.3 Problem 3: Lack of empirical knowledge of developers' communications.

The existing approaches provide limited empirical knowledge of community smell detection through communication channels i.e., sentiment analysis, pull requests, and issue report discussions.

- The communication channels between developers play a crucial role and are key aspects in detecting community smells. Capturing more communication channels between developers leads to providing better detection of community smells. While previous research has shown the feasibility of extracting interactions between developers from online platforms, it has not specifically focused on detecting community smells through these channels, such as sentiment analysis, pull requests, and issue report discussions.

0.3 Research objectives

The main goal of this thesis is to improve the detection of community smells in open-source projects. The sub-objectives are summarized as follows:

Objective 1: Develop an automatic approach that accurately detects community smells and their symptoms in open-source projects and translate the formal definitions and symptoms of community smells into actionable detection rules.

Objective 2: Handle the overlapping symptoms of existing community smells and enhance the community smells detection.

Objective 3: Leverage the rich information provided by the communication's channels, such as sentiment analysis, pull requests, and issue discussions on online platforms to gain a better detection of community smells in open-source projects.

0.4 Main contributions

To overcome the previously identified problems, we propose the following contributions:

Contribution 1: Automating community smells detection.

To automate the detection of community-smells we propose machine learning (ML) approaches. In this contribution, we applied ML to accurately detect community smells and their symptoms in open-source projects. Firstly, we collected a large dataset and provided a comprehensive list of socio-technical metrics for community smell detection. Secondly, we conducted and designed an approach to improve the detection of community smells and investigate the impact of socio-technical characteristics on such smells in open-source software projects. Then, we are presenting an automated approach to implement the proposed framework for detecting community smells in open-source software projects. Furthermore, we have made the tools, approaches, and datasets used in this thesis publicly available to accelerate future research on community smells in the software engineering domain.

Contribution 2: An approach to handle the interleaving symptoms between community smell types.

We introduce an approach based on a set of interleaving organizational-social symptoms that characterize the existence of community smell instances in a software project. We have developed a multi-label learning model to detect several common types of community smells. For this purpose, we have employed the ensemble classifier chain (ECC) model that transforms multi-label problems into several single-label problems. Then, we used genetic programming (GP) to find the optimal detection rules for each smell type.

Contribution 3: A comprehensive framework for improving the detection of community smells.

In the third contribution, we build a comprehensive framework based on new communication channels (pull requests, issue reports, and sentiment analysis). We present a study that highlights the community smells challenges in open source projects. The study introduces a framework for detecting community smells that integrates multiple data sources, including social network graph analysis (commits, pull requests, and issue reports discussions), sentiment analysis, and truck factor metrics. Our framework illustrates the potential benefits in a real-life scenario by asking SE practitioners to give feedback about the community smells in their projects.

0.5 Thesis organization

The thesis consists of three parts and six chapters that are organized as follows: Chapter 2 provides the necessary background related to community smells in software engineering and research related to sentiment analysis. Chapter 3 presents the first part of this thesis's goals "Understanding the challenges of community smells and exploring their attributes" and provides the results of empirical studies related to community smells challenges and attributes. In Chapter 4, we propose an approach that supports the automated detection of community smells. We formulated the community smells detection problem as an optimization problem to find the optimal detection rules for each smell type by using genetic programming (GP). Chapter 5, presents the third part of this thesis, "Improving community smell detection" and show our results for improving community smell detection in open source projects. Finally, Chapter 6 summarizes the thesis and discusses some directions for future work. The whole picture of this thesis is presented in the following figure 0.1:

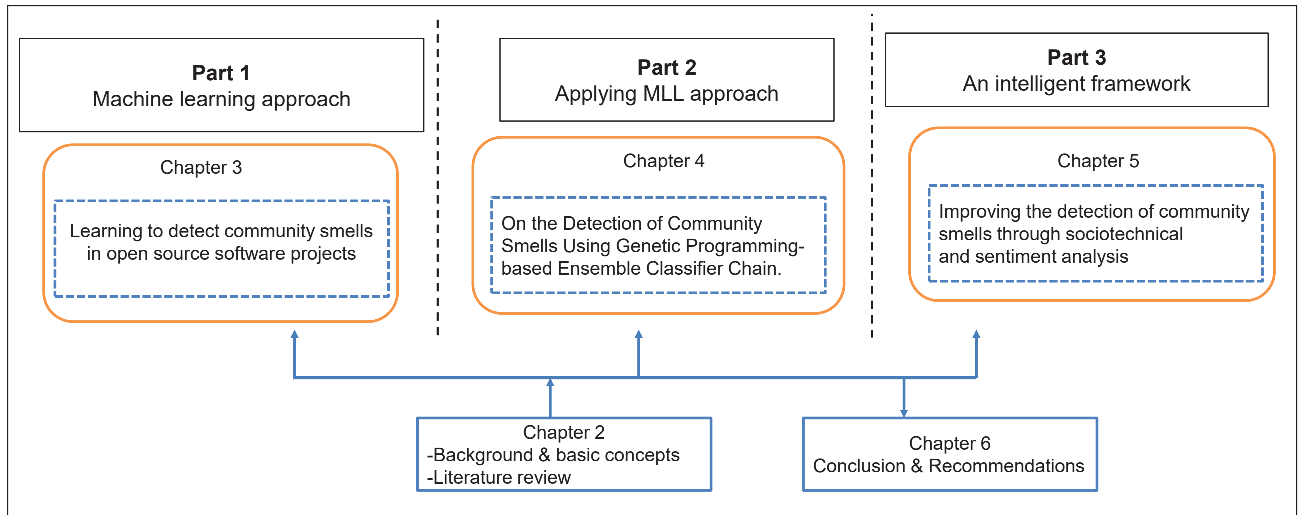


Figure 0.1 Thesis organization

0.6 Publications

The following is a list of our publications related to this dissertation:

- Almarimi, Nuri, Ali Ouni, and Mohamed Wiem Mkaouer. "Learning to detect community smells in open source software projects." Knowledge-Based Systems 204 (2020): 106201.
- Almarimi, Nuri, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. "On the detection of community smells using genetic programming-based ensemble classifier chain." In Proceedings of the 15th International Conference on Global Software Engineering, pp. 43-54. 2020.
- Almarimi, Nuri, et al. "Almarimi, Nuri, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. "Improving the detection of community smells through socio-technical and sentiment analysis." Journal of Software: Evolution and Process (2022): e2505." Journal of Software: Evolution and Process (2022): e2505.
- Almarimi, Nuri, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. "csDetector: an open source tool for community smells detection." In Proceedings of the 29th ACM Joint

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1560-1564. 2021.

CHAPTER 1

STATE OF THE ART

1.1 Introduction

In this chapter, we present a literature review that is connected to the research work presented in this thesis. We begin by providing essential background information necessary for comprehending the thesis. Following that, we survey the relevant research that relates to the primary themes of this research work.

1.2 Background

In this section, we provide the necessary background for Social debt and community smells, developer's social networks, and search-based software techniques.

1.2.1 Social debt and community smells

Social debt is connected to poor organization structures that often lead to short and/or long-term social issues within a project (Tamburri, Kruchten, Lago & van Vliet (2013a)). These problems generate unforeseen additional costs in a software development environment, where non-optimal or uniform socio-technical decisions influence both the social and technical aspects of the software development communities. Moreover, such issues are not easily detectable and visible as well as can be postponed in time (Tamburri *et al.* (2013a, 2015b)).

Much like the concept of technical debt, social debt impacts heavily on software development success, and over time this debt will continue to accumulate interest and is not repaid until the tasks are completed, which creates problems in the long term and makes it harder to implement changes in the future (Tamburri *et al.* (2015b)). The social debt is similar to the technical debt while decisions in technical debt are about technologies and their applications, the decisions that cause social debt are about people themselves and their interactions (Tamburri *et al.* (2015b)).

In the same meaning of the code smells concept, the social debt researchers defined "Community smells" as social related anti-patterns useful to understand negative characteristics and trends (Tamburri *et al.* (2019c); Caballero-Espinosa, Carver & Stowers (2022)). Community smells are defined as "socio-technical anti-patterns that may appear normal but in fact, reflect unlikable community characteristics" (Tamburri, Palomba & Kazman (2021)). Thus, community smells are useful to understand negative organizational and social aspects and developer behaviors within a community. In other words, community smells are a set of social and organizational circumstance that happened over time with implicit additional cost to a project, the consequence of this problem cause social dept in the form of delays, mistrust, uninformed and miscommunication architectural decision-making (Tamburri *et al.* (2021, 2019c)).

Tamburri et al. identified several community smells by analyzing social characteristics and technical metrics in open-source projects, which may affect the health of OSS communities Tamburri *et al.* (2016). As an extension of their research, Tamburri et al. introduced a tool called YOSHI that automates the discovery of community patterns in open-source projects Tamburri, Palomba, Serebrenik & Zaidman (2018). This tool characterizes open software projects based on measurable attributes and formal detection rules for community factors. Siemens also introduced a tool called CodeFace that builds social developer networks based on community maps Joblin, Mauerer, Apel, Siegmund & Riehle (2015a). Later, Tamburri et al. Tamburri *et al.* (2021) built upon CodeFace4smell scenarios to detect various types of smells in OSS projects, using statistical metrics from the GitHub repository and mailing lists among developers Tamburri, Palomba & Kazman (2019). The tool tracks the history of a repository and the mailing lists of projects to build social networks of developers.

1.2.2 Developer social networks

In a software system community could be formed the interactions and relationships between developers as a self-organized network, which can be considered as a developer social network (Hong, Kim, Cheung & Bird (2011)). Based on the developer network concept a socio-technical developer network can be generated from socio-technical connections which present

the collaboration and communication channels. The researchers have created developer social networks from every possible development source information to allow the usage of social network analysis methodologies and metrics. Such social networks can be generated from code source history and mostly from other open and accessible data sources used within software development such as bug reporting, pull requests, and version control systems (Lopez-Fernandez, Robles, Gonzalez-Barahona *et al.* (2004)).

Social network analysis could be presented by actors and how they are related to each other through relationships. Such a relationship could be extracted by mining software development code and the consequences of this mining that able to study all the possible ways in which people interact through all the available information sources used to develop software. Recently studies in empirical software engineering apply social network analysis methodologies since they have a solid systematic and quantitative framework (Meneely & Williams (2011a)). A social network and its actors have main two properties connectivity and distance (Lin & Chen (2004)). Connectivity could be measured using metrics density, size, centrality, and reachability of the social network, it could be defined as a member of the social network is more power on a network where this developer is more connected and can be considered more influential in the community. Distance in a social network represents by the closeness metrics where the closeness of two actors within the network may identify properties differences like distribution and consistency. Connection and distance are the main characteristics to enable the identification of sub-communities within a social network, which are defined as “subsets of actors among whom there are relatively strong, direct, intense, frequent, or positive ties” (Lin & Chen (2004)).

With such resource information, the researchers can easily and freely analyze defects, communications, and distributed collection habits of open-source software developers. The communication and coordination activities between developers are public and accessible to anyone and this allows researchers to track coordination and communication activities and mine them through the usage of developer social networks (Bird, Gourley, Devanbu, Gertz & Swaminathan (2006)).

1.2.3 Sentiment Analysis for Software Engineering

Sentiment analysis has been used widely in software engineering. It has been applied to many tasks across different software engineering artifacts, including technical elements such as commit messages and issues, and crowd content such as forum messages and user reviews (Lin *et al.* (2018)). Developers frequently express sentiment in the commit messages and issues in open source projects i.e., GitHub (Jurado & Rodriguez (2015)). In their study, (Guzman, Azócar & Li (2014)) analyzed the sentiment of commit comments in GitHub and found that projects with more distributed teams generally exhibit more positive emotional content, while comments written on Mondays tend to express more negative emotions. Sinha (Sinha, Lazar & Sharif (2016)) conducted a similar study, analyzing 28,466 projects over a seven-year year time-frame., and found that the majority of sentiment expressed was neutral, with Tuesdays having the most negative sentiment overall. They also identified a strong positive correlation between the number of files changed and the sentiment expressed in the associated commits. Ortu (Ortu *et al.* (2015a)) analyzed 560,000 JIRA comments and found that positive sentiment expressed in the issue description may improve issue-fixing time. Finally, Souza (Souza & Silva (2017)) explored the relationship between developers' sentiment and continuous integration server builds and discovered that negative sentiment can both impact and be impacted by the build process results.

Sentiment analysis has been employed to detect the emotional state of developers due to the potential impact of emotions on their productivity, job satisfaction, and task completion quality (Rousinopoulos, Robles & González-Barahona (2014)). Guzman and Bruegge (Guzman & Bruegge (2013a)) leveraged sentiment analysis to explore the significance of emotional awareness in development teams, while Gachechiladze and colleagues (Gachechiladze, Lanubile, Novielli & Serebrenik (2017)) used sentiment analysis to create a detailed model for detecting anger. Additionally, the study by Pletea and colleagues (Pletea, Vasilescu & Serebrenik (2014)) presented evidence that developers have a tendency to express negativity when discussing security-related topics. Finally, Garcia and colleagues (Garcia, Zanetti & Schweitzer (2013)) examined the relationship between emotions and contributor activity in the GENTOO Open Source Software project and determined that contributors are more likely to become inactive

when they exhibit strong positive or negative emotions in the issue tracker or deviate from the anticipated range of emotions in the mailing list.

Several research works have recently been conducted on various social and sentiment polarity analyses in open-source projects. Guzman et al. performed a sentiment polarity analysis of commit comments in GitHub and provided evidence of a correlation between negative sentiment and commit activity performed on Mondays in 29 GitHub projects (Guzman *et al.* (2014).). Pletea et al. (2014) studied GitHub discussions related to security, extracted from discussions around commits and pull requests, and found that more negative emotions are expressed in security-related discussions compared to other discussions Pletea *et al.* (2014). Murgia et al. (2014) manually analyzed whether development discussions, such as bug repositories, carry emotional information about software development. The results of their study indicate that issue reports do express emotions towards design choices, maintenance activity, or colleagues Murgia, Tourani, Adams & Ortu (2014). Guzman and Bruegge (2013) proposed a sentiment analysis approach for discussions in different platforms, such as mailing lists and web-based software collaboration tools like Confluence, to enhance emotional awareness in software development teams. They analyzed the sentiment emotions expressed in commit comments with respect to different factors, such as programming language and different periods of time, and found that negative emotions appear more frequently in projects developed in Java and in commit comments written on Mondays Guzman & Bruegge (2013b). Ortu et al. (2015) empirically analyzed the relationship between sentiment, emotions, and politeness of developers in over 560K comments of the Jira issue tracking system. They found that the shorter the issue-fixing time, the more likely emotional expressions such as JOY and LOVE are found in the comments Ortu *et al.* (2015b). Similarly, Gachechiladze et al. developed an approach to detect anger emotions in issue reports and built a classifier capable of analyzing emotions expressed in developer communication Gachechiladze *et al.* (2017). More recently, Islam and Zibran (2016) studied the impacts of emotions on software artifacts, specifically commit messages, and explored the potential for exploiting emotional variations in software engineering activities. They found that while the polarities of developers' emotions vary significantly depending on the type of tasks

they are engaged in, the majority (65%) of commit messages are neutral in emotion, with positive emotions accounting for 13% of the commit comments and negative emotions accounting for 22% Islam & Zibran (2016).

1.2.4 Search-based software engineering (SBSE)

The term "SBSE" was initially introduced by Harman and Jones in 2001, who defined it as the use of search-based approaches to tackle optimization problems in software engineering (Harman & Jones (2001)). The fundamental concept behind SBSE is to transform software engineering problems into search problems, by defining them based on solution representation, fitness function, and solution change operators. Once a software engineering problem is reformulated as a search problem, a range of metaheuristic techniques can be employed to identify near-optimal solutions to the problem.

Indeed, in recent few years, many SBSE approaches have been applied to a wide variety of software-engineering problems, including involving single to many-objective techniques that can be used to solve the problem (Harman, Mansouri & Zhang (2012b); Ouni, Kessentini, Sahraoui, Inoue & Deb (2016a); Ouni, Kula & Inoue (2016b); Ouni, Kessentini & Sahraoui (2013a); Ouni, Kessentini, Sahraoui & Hamdi (2013c)). We will investigate in this thesis the use of SBSE techniques for automating the detection of community smells. More specifically, we used multi-objective metaheuristic techniques Genetic Programming (MOGP). Genetic Programming (GP) (Koza (1992)) is a powerful heuristic search optimization method. The basic idea is to explore the search space by making a population of candidate solutions, also called individuals, evolve toward a “good” solution to a specific problem.

1.3 Related Work

This section presents the related work that is relevant to the main themes of this research work. In particular, the related work can be divided broadly into three research areas: (1) Open source community’s health, (2) Community smells detection, and (3) Sentiment analysis detection.

1.3.1 Open source community's health

Recently many papers showed that community health can impact software quality (Catolino, Palomba & Tamburri (2019a); Palomba *et al.* (2018b,a)). In their study, Tamburri, Palomba, and Kazman (Tamburri, Palomba & Kazman (2020)) conducted a thorough and extensive review of the literature and developed a grounded theory comprising of more than 500 factors that contribute to both success and failure. These factors were divided into 14 distinct clusters, which were manually validated.

In open source projects, several socio-technical metrics can be found to determine community's health i.e., stickiness and magnetism (Yamashita, Kamei, McIntosh, Hassan & Ubayashi (2016); Yamashita, McIntosh, Kamei & Ubayashi (2014)), and turnover and communicability (Palomba & Tamburri (2021)).

Cataldo et al. (Cataldo, Herbsleb & Carley (2008)) utilized the idea of congruence to explore the correlations between various technical and work-related dependencies among software developers, and how these dependencies affect their productivity. Similarly, Tamburri et al. (Tamburri *et al.* (2019b))discovered that the socio-technical congruence, as defined by Cataldo et al. (Cataldo *et al.* (2008)), is linked to a reduced number of community smells, making it a useful tool for monitoring community health. Another study have also identified health metrics associated with community smells (Palomba & Tamburri (2021)). Catolino et al. (Catolino, Palomba, Tamburri & Serebrenik (2021)), for instance, identified community health metrics for four specific community smells.

According to Crowston and Howison (Crowston & Howison (2006)), a healthy open-source community has a hierarchical structure with core developers and leaders at the center, surrounded by distinct layers of codevelopers, active users, and passive users. The authors suggest that the health of a community can be identified by how it deals with challenging tasks, and healthy communities should address these issues openly. Jansen offers a framework to assess the health of an open-source ecosystem at a higher level, beyond individual projects (Jansen (2014)). Goggins, Lumbard, and Germonprez (Goggins, Lumbard & Germonprez (2021)) conducted

an extensive review of the current methods used for analyzing project health and identified their limitations. They found that repository histories are often compressed, and there is little consideration given to changes over time when making inferences. Moreover, they noted that activity is frequently used as a proxy for assessing project health, but this is inadequate in open-source projects as studies are often conducted at a small unit level, such as a software commit. For instance, Onoue et al. (Onoue, Kula, Hata & Matsumoto (2017)) used workforce and gross product pull requests as activity indicators to determine project health, and Xia et al. (Xia, Fu, Shu & Menzies (2020)) developed predictors for seven project health indicators, including the number of commits and closed issues, but these indicators are all limited to project activity.

Prior work focused on community health and its impact on software quality using socio-technical metrics. However, these works achieved a limited list of socio-technical metrics as well as community smells types. One of the main objectives of this thesis is to conduct empirical studies to explore new community smells that may impact community health.

1.3.2 Community smells detection approaches and tools

Community smells could be defined as sets of social-organizational structures and circumstances, which may lead to social debt (Tamburri *et al.* (2015b)). Developers consider community smells significant risks to community health (Tamburri *et al.* (2019b)). Many approaches and tools have been proposed to detect community smells in open-source projects. Tamburri et al. (Tamburri *et al.* (2019b)) proposed the CodeFace4Smells tool, as an automated approach that used social network metrics to detect four types of community smells. Palomba and Tamburri (Palomba & Tamburri (2021)) conducted a study where they investigated the ability of socio-technical metrics to predict community smells. They developed a model that uses machine learning that uses a random forest algorithm for this purpose. Avelino et al. (Avelino, Passos, Hora & Valente (2016b)) proposed Truck Factor, a specialized tool for identifying the number of team members whose absence or quit (hit by a truck) before the project is completed, would cause a significant negative impact on a project. A low truck factor is considered a community

smell as it can result in a loss of critical knowledge when developers leave the team. In another study, Paradis and Kazman (Paradis & Kazman (2021)) introduced Kaiaulu, an API that analyzes different sources of software development data, including git logs, mailing lists, and files, among others. Kaiaulu facilitates data interoperability through author and file linkage, filters, and popular code metrics. It can identify three types of community smells using the same detection rules as CodeFace4Smells but employs a more robust community detection algorithm. Recently, the developers introduced a CADOCS tool (Voria *et al.* (2022)) a client-server conversational agent that builds on top of a previous community smell detection tool csDETECTOR (Almarimi, Ouni, Chouchen & Mkaouer (2021)) and makes it available as a readily-usable SLACK bot.

Several studies have found correlations between community smells and other factors. For instance, Catolino et al. (Catolino, Palomba, Tamburri, Serebrenik & Ferrucci (2019c,b)) investigated the relationship between community smells and team composition, specifically how gender diversity affects the occurrence of community smells. Tamburri et al. (Tamburri, Kazman & Van den Heuvel (2019a)) examined the co-occurrence of community smells with software architecture smells. Palomba et al. (Palomba *et al.* (2018a)) provided evidence that code- and community smells in software engineering are interrelated. In their subsequent work (Palomba *et al.* (2018b)), they concluded that community-related factors contribute to the intensity of code smells.

Common to these approaches and tools is that they propose techniques to detect and investigate the community smells. They mostly devise a limited and generic list of characteristics and symptoms to generate smells detection rules that characterize community smells. One of the main goals of this thesis aims at learning from existing smells to detect new ones to help developers better allocate their resources and save time and efforts through automated detection of smells.

1.3.3 Sentiment analysis detection approaches and tools

In recent years, there has been an increase in the utilization of techniques and tools for extracting opinions from online platforms through automated means. (Lin *et al.* (2022); Lin *et al.* (2018)). The sentiment analysis tools have been widely adopted by the software engineering community for diverse purposes. These tools have been utilized to evaluate the polarity of app reviews (Goul, Marjanovic, Baxley & Vizecky (2012); Panichella *et al.* (2015)), identify negative opinions regarding APIs (Zhang & Hou (2013)), detect happiness or distress within a development team (Tourani, Jiang & Adams (2014)), and determine the probability of inactivity among developers who express strong emotions in issue trackers (Garcia *et al.* (2013)). Additionally, researchers have studied the impact of sentiment expressed on issues' comments and the issue resolution time (Ortu *et al.* (2015a)), as well as the sentiment of developers' commits (Sinha *et al.* (2016)).

There are many sentiment analysis tools applied to software engineering applications. The most widely used sentiment analysis tool is SentiStrength (Thelwall, Buckley, Paltoglou, Cai & Kappas (2010)), which was initially trained on MySpace5 comments. SentiStrength's foundation is based on a sentiment word strength list, which is made up of 298 positives and 465 negative terms with an associated positive or negative strength value. Additionally, SentiStrength utilizes a spelling correction algorithm and other word lists, such as a booster word list and a negating word list, to improve the sentiment assessment. The sentiment score of each word in a sentence under analysis is determined by SentiStrength, which sums up the individual scores to derive the sentence's overall sentiment. SentiStrength's straightforward methodology allows for customization to a specific context by establishing a list of domain-specific terms with corresponding sentiment scores.

NLTK (Tausczik & Pennebaker (2010)) is a sentiment analysis tool based on a lexicon and rule system, with VADER (Valence Aware Dictionary and sentiment Reasoner) as its primary component. VADER is uniquely designed to analyze social media texts and incorporates a "gold-standard" sentiment lexicon extracted from microblog-like contexts that have been manually validated by multiple independent human judges. Stanford CoreNLP (Danescu-

Niculescu-Mizil, Sudhof, Jurafsky, Leskovec & Potts (2013b)) utilizes a Recursive Neural Network as its foundation, which distinguishes it from SentiStrength and NLTK in its capability to assess a sentence's sentiment based on how words contribute to the sentence's meaning, rather than simply adding up the sentiment of each individual word. It was trained using movie reviews.

Werder and Brinkkemper (2018) developed MEME, a tool that extracts emotions from software engineering text based on data sources from GHtorrent and GitHub, and achieved better performance compared to state-of-the-art tools, such as the Syuzhet R package for emotion analysis Werder & Brinkkemper (2018). Recently, Murgia et al. (2018) analyzed emotional information in issue reports and found that developers express emotions such as gratitude, joy, and sadness. Based on their results, the authors advocated for the feasibility of a machine learning classifier to identify issue comments containing gratitude, joy, and sadness Murgia, Ortu, Tourani, Adams & Demeyer (2018). Lin et al. (2018) reported their experience in building a software library recommender by exploiting developers' opinions mined from Stack Overflow using various sentiment analysis tools. They highlighted issues with the accuracy of existing sentiment analysis tools found in the literature Lin *et al.* (2018). Later, El Asri et al. (2019) investigated perceived sentiments during code review and found that contributors frequently express positive and negative sentiments during code review activities, depending on their position within the social network of the reviewers, such as core vs. peripheral contributors El Asri, Kerzazi, Uddin, Khomh & Idrissi (2019). More recently, Raman et al. (2020) attempted to find and understand unhealthy interactions in software development teams by developing and demonstrating a measurement instrument to detect toxic discussions in GitHub issues. They used an SVM classifier to analyze trends over time and in different GitHub communities and found that toxicity varies by community and tends to decrease over time Raman, Cao, Tsvetkov, Kästner & Vasilescu (2020).

While previous research has demonstrated the feasibility of extracting sentiment analysis from online platforms, it has not specifically focused on identifying community smells through sentiment analysis. The main objective of this thesis is to empirically validate the challenges associated with identifying community smells using sentiment analysis.

1.4 Chapter Summary

This chapter presented background about the community smells and their relevant research. Then, it surveys previous work on community smells in the software engineering domain. Next, it presents prior research on using tools and approaches to detect community smells in the software engineering domain. While the existing approaches attempt mainly to characterize and analyze organizational social structures in software development communities, they do not cover multi-perspective characteristics of software projects. They mostly devise a limited and generic list of characteristics and symptoms to generate smell detection rules that characterize community smells. However, such rules will need substantial human effort and expertise to calibrate these rules for each smell type and adapt them to different projects, organizations, and contexts. Hence, we believe that an appropriate detection is needed to fill this gap. In the next chapters, we describe our empirical study to identify and explore community smells detection challenges. Also, we introduce our proposed tool to help developers in the early detection of community smells.

CHAPTER 2

LEARNING TO DETECT COMMUNITY SMELLS IN OPEN SOURCE SOFTWARE PROJECTS

Nuri Almarimi[a] , Ali Ouni[a] , Mohamed Wiem Mkaouer[b]

[a] Department of Software Engineering and IT, École de Technologie Supérieure
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

[b] Rochester Institute of Technology, 1 Lomb Memorial Dr, Rochester,
NY 14623, United States

Paper published in *knowledge-based systems journal*, July 2020.

Abstract

Community smells are symptoms of organizational and social issues within the software development community that often lead to additional project costs. Recent studies identified a variety of community smells and defined them as sub-optimal patterns connected to organizational-social structures in the software development community. To early detect and discover existence of potential community smells in a software project, we introduce, in this paper, a novel machine learning-based detection approach, named csDetector, that learns from various existing bad community development practices to provide automated support in detecting such community smells. In particular, our approach learns from a set of organizational-social symptoms that characterize the existence of potential instances of community smells in a software project. We built a detection model using Decision Tree by adopting the C4.5 classifier to identify eight commonly occurring community smells in software projects. To evaluate the performance of our approach, we conduct an empirical study on a benchmark of 74 open source projects from Github. Our statistical results show a high performance of csDetector, achieving an average accuracy of 96% and AUC of 0.94. Moreover, our results indicate that the csDetector outperforms two recent state-of-the-art techniques in terms of detection accuracy. Finally, we investigate the most influential community-related metrics to identify each community smell type. We found that the number of commits and developers per time zone, the number of developers per community,

and the social network betweenness and closeness centrality are the most influential community characteristics.

Keywords. Community smells detection , Social debt, Socio-technical metrics, Machine learning.

2.1 Introduction

Software engineering can be primarily described as an organized effort of “social" activity of organizations, individuals, and stakeholders to build a software product. Therefore, the organizational social structure in a software development community, including the interactions among developers, is an essential prerequisite for a successful software product.

An intrinsic characteristic of software projects resides in their (rapid) evolution involving a large number of clientele and stakeholders (Lehman (1980)). Although such growth is beneficial for the project’s success, there reveals another set of challenges that go beyond maintaining the software quality and functionality, and that impact the socio-managerial structure of the project. Hence, several researchers and practitioners revealed how critical is the evolution of organizational aspects to prevent it from decay, and consequently software projects failures (Navarro (2001); Avelino *et al.* (2016b)). Recent studies coined the set of socio-technical patterns, negatively impact the organizational health of the project, as community smells (Tamburri *et al.* (2013b)). Community smells are connected to circumstances based on poor organizational and social practices that lead to the occurrence of social debt (Tamburri *et al.* (2016, 2015b)). Social debt is connected to poor organization structures that often lead to short and/or long term social issues within a project. These problems could manifest in several forms, e.g., lack of communications or coordination among members in a software community. Indeed, M. Scott Peck pointed out that: *"There can be no vulnerability without risk; there can be no community without vulnerability; there can be no peace, and ultimately no life, without community" Peck (2002).*

For example, one of the common community smells, is the "organizational silo effect" (Palomba *et al.* (2019); Tamburri, Lago & Vliet (2013c); Bindrees, Pooley, Ibrahim & Bental (2014)) which manifests as a recurring social network sub-structure featuring highly decoupled developers community structures. From an analytical perspective, this community smell can be seen as a set of patterns over a social network graph and could be detectable using different graph connectivity measurements.

To early detect potential instances of poor community practices during a software project, efficient and automated techniques are needed (Tamburri *et al.* (2016, 2015b)). Although there have been a few studies to define, characterize, and detect community smells in open source projects, they are applied, in general, to a limited set of projects, and their generalizability requires a manual effort and human expertise to define and calibrate a set of detection rules to match the symptoms of a community smell with the actual characteristics of a given software project (Avelino *et al.* (2016b); Palomba *et al.* (2018b); Tamburri *et al.* (2019c,b)).

We build on top of previous studies to develop a model that learns from previously detected community smells, by considering the patterns of their characteristics into features, which allows their classification. In this paper, we introduce a novel machine learning based approach for community smell detection, named csDetector. The proposed approach learns from a set of organizational-social symptoms that characterize the existence of a potential community smell. csDetector adopts a detection model to provide a learn-by-example process to automatically detect community smells from various software projects. The proposed approach allows practitioners to choose project examples that are mostly similar to their context, as it learns from their associated community smells, to provide a personalized detection model for their project.

To evaluate our approach, we experiment several existing machine learning algorithms, including C4.5, JRip, Random Forest, Nave Bayes, SMO, and LibSVM, on eight commonly occurring community smell types (Tamburri *et al.* (2015b); Avelino *et al.* (2016b)) extracted from 74 open-source software systems. Results show that the experimented techniques can provide high performance for all the considered community smells. However, the highest performance was

achieved by the C4.5 classifier. In particular, the obtained results indicate that C4.5 is able to detect the different types of community smells with an average accuracy of 96%, an F-measure of 92% and an Area Under the receiver operating characteristic Curve (AUC) of 94%. The main contributions of the paper can be summarized as follows:

- We introduce a novel approach, named csDetector, for community smells detection using decision tree to learn from a set of organizational-social symptoms that characterize the existence of potential instances of community smellsin active real word open source software projects.
- We report the results of an empirical study with an implementation of our approach on a benchmark of 74 open source projects from Github and eight common community smells. Results show a high performance of the proposed approach, achieving an average accuracy of 96% and AUC of 0.94. Moreover, we found that the C4.5 decision tree algorithm outperforms 5 widely used machine learning algorithms including JRip, random forest, Naïve Bayes, SMO, and LibSVP. Moreover, the statistical analysis of the obtained results show that our approach outperforms two recent state-of-the-art techniques in terms of detection accuracy.
- We conduct a set of experiments to investigate the most influential community-related metrics to identify each community smell type. We found that the number of commits and developers per time zone, the number of developers per community, and the social network betweenness and closeness centrality are the most influential community characteristics. We also conduct a sensitivity analysis to assess if there exists influential data points than may influence the stability of our model.
- We provide our comprehensive dataset collected and used in this study publicly available for replication purposes, and to foster research in the field of community smells and social debt (Almarimi (2019)).

Replication package. We provide our comprehensive dataset collected and used in this study publicly available for replication purposes.

Paper organization. The remainder of the paper is organized as follows. Section 2 provides the necessary background on community smells. In Section 3, we describe our approach for

community smells detection. In Section 4, we present our empirical evaluation setup, and then present and discuss the obtained results in Section 4. We discuss, in Section 6, the threats to validity. Finally, in Section 7, we conclude and outline our future work.

2.2 Background

In this section, we present the necessary background related to community smells. Then, we discuss the related work that translated the definition of community smells into actionable detection rules.

2.2.1 Community smells definitions

Community smells are defined as a set of social-organizational circumstances that occur within the software development community, and that have a negative effect on the relations health within the development community and may cause social debt over time (Tamburri *et al.* (2015b)). In our work, we investigate the detection of common community smells that are identified and defined in the literature. We refer to following list of existing community smell types (Tamburri *et al.* (2015b); Avelino *et al.* (2016b)).

- **Organizational Silo Effect (OSE):** The Organizational Silo Effect smell is manifested when a too high decoupling between developers, isolated subgroups, and lack of communication and collaboration between community developers occur. The consequence of this smell is an extra unforeseen cost to a project by wasted resources (e.g., time), as well as duplication of code (Tamburri *et al.* (2016, 2015b)).
- **Black-cloud Effect (BCE):** The black-cloud effect smell occurs when developers have a lack of information due to limited knowledge sharing opportunities (e.g., collaborations, discussions, daily stand-ups, etc.), as well as a lack of expert members in the project that are able to cover the experience or knowledge gap of a community. The BCE smell may cause mistrust between members and creates selfish behavioral attitudes (Tamburri *et al.* (2015b)).
- **Prima-donnas Effect (PDE):** The prima-donnas effect smell occurs when a team of people is unwilling to respect external changes from other team members due to inefficiently structured

collaboration within a community. The presence of this smell may create isolation problems, superiority, constant disagreement, uncooperativeness and raise selfish team behavior, also called “prima-donnas” (Tamburri *et al.* (2016, 2015b)).

- **Sharing Villainy (SV):** This smell is caused by a lack of high-quality information exchange activities (e.g., face-toface meetings). The main side effect of this smell limitation is that community members share essential knowledge such as outdated, wrong and unconfirmed information (Tamburri *et al.* (2016, 2015b)).
- **Organizational Skirmish (OS):** This community smell is caused by a misalignment between different expertise levels and communication channels among development units or individuals involved in the project. The existence of this smell leads often to dropped productivity and affects the project’s timeline and cost (Tamburri *et al.* (2015b)).
- **Solution Defiance (SD):** The solution defiance smell occurs when the development community presents different levels of cultural and experience background, and these variances lead to the division of the community into similar subgroups with completely conflicting opinions concerning technical or socio-technical decisions to be taken. The existence of the SD smell often leads to unexpected project delays and uncooperative behaviors among the developers (Tamburri *et al.* (2015b)).
- **Radio Silence (RS):** The radio silence smell occurs when a high formality of regular procedures takes place due to the inefficient structural organization of a community. The RS community smell typically causes changes to be retarded, as well as a valuable time to be lost due to complex and rigid formal procedures. The main effect of this smell is an unexpected massive delay in the decision-making process due to the required formal actions needed (Tamburri *et al.* (2015b)).
- **Truck Factor Smell (TFS):** The truck factor smell occurs when most of the project information and knowledge are concentrated in one or few developers. The presence of this smell eventually leads to a significant knowledge loss due to the turnover of developers (Avelino *et al.* (2016b)).

In this paper, we focus primarily on the above-mentioned smells that are frequently occurring in the software industry (Tamburri *et al.* (2016, 2015b, 2019b); Avelino *et al.* (2016b)). We selected these smells as they are defined and investigated in recent studies and known to have negative effects in practice, which may lead to social debt (Tamburri *et al.* (2013a)) and poor source code quality (Tamburri *et al.* (2021)).

2.3 csDetector: Community smells detection using machine learning

In this section, we present our approach, csDetector, for detecting community smells. csDetector aims at detecting various social-technical issues that can take place in open source projects. Figure 2.1 presents an overview of our proposed approach which consists of three main components (1) training data collection and organizational-social metrics extraction, (2) training model building using J48 classifier, and (3) community smells detection. In the following, we explain each of these components.

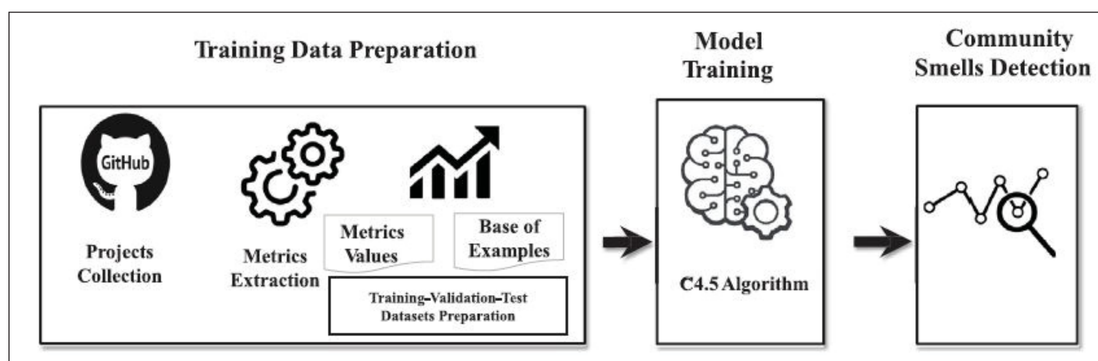


Figure 2.1 An overview of the proposed csDetector approach.

2.3.1 Data collection

To build a base of real-world community smell examples that occur in software projects, we select a set of software projects which are diverse in nature (*e.g.*, size, application domain, popularity, etc.) that are most likely to experience community smells. Additionally, we considered open-source projects to access to their development history to calculate their social-organizational

characteristics. We selected a set of projects from github that exhibit various characteristics by considering the following criteria :

- *Commit size*: the projects vary from medium size >10 KLOC, large size (10- 60) to very large size < 60 KLOC.
- *Community size*: the projects team size vary from medium <100 members, large (100-900) members, to very large > 900 members.
- *Programming language*: the selected projects are implemented in different programming languages including Java, C#, Python and C.
- *Existence of community smells:* the project contains at least one community smell.

From the obtained projects we randomly selected among them. We, then, manually inspected all of these projects to identify the potential existence of community smells using a crawler to collect and analyze their change history based with the assistance of existing guidelines from the literature (Tamburri *et al.* (2015b, 2013c,b, 2016, 2019b); Avelino *et al.* (2016b)) to summarize and visualize different aspects, e.g., social networks, metrics variations, developers commits contributions and collaborations, etc. Each author independently identified potential community smells. In case of a conflict about a smell, i.e., there is a disagreement about the presence of a smell, the authors attempt to resolve it through an open discussion to reach an agreement about the salient symptoms of the candidate smell. All projects for which there was no agreement about a candidate community smell, were excluded from our ground truth dataset. We finally ended up with 74 projects from an initial list of 107 projects that have diverse types of community smells (Almarimi (2019)). The details of the collected dataset are reported in Table 4. Each system repository can have one or many projects. Each project is one data point, so the total number of data points is 74 for each smell type. The dependent variable is a specific smell type. Moreover, Table 2.1 and Fig. 2.2 provides more statistical details about our analyzed dataset in terms of the distribution of smells, developers, commits and days lived.

Table 2.1 Dataset statistics.

Data	Statistic
Number of systems	23
Number of projects	74
Number of projects having at least one smell	74
Total number of smells	236
Average number of smells per project	3.18
Number of projects with <50 developers	30
Number of projects with 50 − 150 developers	21
Number of projects with >150 developers	23
Average number of commits per project	1103
Average number of days in each project	3233

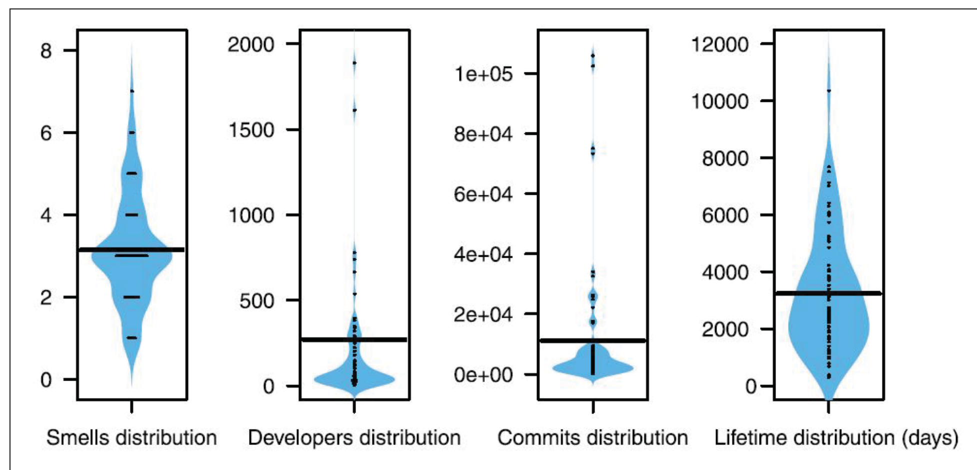


Figure 2.2 Beanplots for the distribution of smells, developers, commits, and lifetime among the studied projects.

2.3.2 Metrics framework

To capture community smells symptoms, we rely on (1) et set of metrics defined in previous studies (Avelino *et al.* (2016b); Tamburri *et al.* (2019b); Nagappan, Murphy & Basili (2008b); Pinzger, Nagappan & Murphy (2008b); Nordio *et al.* (2011)). and (2) a new set of metrics that we introduce to capture more community-related proprieties that can be mined from the projects history. These metrics analyze different aspects in software development communities including organizational dimensional, social network characteristics, community developer's collaborations, and truck numbers. Table 2.2 depicts our list of metrics as well as state-of-the-art ones used in this research study. Our proposed metrics extend existing metrics to provide more project characteristics generalizations including developer social network, community structures,

geographic dispersion, and developer network formality. For instance, the geographic dispersion metrics, *e.g.*, the *average number of commits per time zone* and the *average number of developers per time zone* would provide a whole view of the distribution of commits and developers per time zone.

Table 2.2 Metrics framework.

Dimension	**Acronym**	**Definition**	**Ref.**
Developer Social Network metrics	NoD	*Number of developers (NoD)*: is the total number of Developers who have changed the code in a project. The more developers modify the same files, the higher the need for collaboration between developers.	Nagappan *et al.* (2008b)
	TAP	*Number of active days of an author on a Project (TAP)*: is ratio of the total number of active days for each developer with respect to a project's lifetime by the total number of developers in a project.	New
	LCP	*Number of changed lines of a code per author in a project (LCP)*: is the total number of times that the code has been changed by a developer with respect to the total number of lines of code and the total number of developers in a project.	Nagappan *et al.* (2008b)
	CD	*Number of core developers (CD)*: is the total number of core developers in a project. A developer is considered as a core community member if he has a high degree of centrality of more than > 0.5 within the developer's social network.	Tamburri *et al.* (2021)
	RCD	*Ratio of core developers (RCD)*: is the ratio of the number of core developers with respect to the total number of developers in a project.	Tamburri *et al.* (2021)
	SD	*Number of sponsored developers (SD)*: is the total number of sponsored developers in a project. We consider a developer that hold a sponsored status if at least 95% of her/his commits are executed during weekdays and the working day time between 8am and 5pm.	Tamburri *et al.* (2021)
	RSD	*Ratio of sponsored developers (RSD)*: is the ratio of the number of sponsored developers by the total number of developers in a project.	Tamburri *et al.* (2021)
Social Network Analysis metrics	DC	*Graph Degree centrality (DC)*: is the number of connections that a developer has. The more connections with others a developer has, the more important the developer is.	Pinzger *et al.* (2008b)
	BC	*Graph Betweenness centrality (BC)*: is a measure of the information flow from one developer to another and devised as a general measure of social network centrality. It represents the degree to which developers stand between each other. A developer with higher BC would have more control over the community as more information will pass through her/him.	Pinzger *et al.* (2008b)
	CC	*Graph Closeness centrality (CC)*: is a measure of the distance between a developer to other developers in the network. This metric is strongly influenced by the degree of connectivity of a network.	Pinzger *et al.* (2008b)
	ND	*Network Density (ND)*: is a measure of a social network as a dense or sparse graph.	Tamburri *et al.* (2021)
Community metrics	NC	*Number of communities (NC)*: is the total number of communities in a project and a measure of the strength of the structure of a project community.	Tamburri *et al.* (2021)
	RCC	*Ratio of commits per community (RCC)*: is the ratio of the number of commits assigned to each community with respect to the total number of commits in a project. It provides a view of the distribution of the commit per community.	New
	RDC	*Ratio of developers per community (RDC)*: is the ratio of the number of developers assigned to each community in a project with respect to the total number of developers in a project. .	New
Geographic Dispersion metrics	TZ	*Number of time zones (TZ)*: is the total number of different time zones of developers in a project.	Nordio *et al.* (2011)
	RCZ	*Ratio of commits per time zone (RCZ)*: is the ratio of the number of commits in each time zone by the total number of time zones in a project. It provides a view of the distribution of the commit per time zone.	New
	RDZ	*Ratio of developers per time zone (RDZ)*: is the ratio of the number of developers per time zone in a project by the total number of time zones in a project. It provides a view of the distribution of the developers per time zone.	New
Formality metrics	NR	*Number of Releases in a project (NR)*: is the total number of releases present in a project.	New
	RCR	*Ratio of Commits per Release (RCR)*: is the ratio of the number of commits assigned to each release with respect to the total number of releases in a project.	New
	FN	*Formal network (FN)*: is calculating by the ratio of milestones assigned to the project and lifetime of the project.	Cataldo, Mockus, Roberts & Herbsleb (2009)
Truck Number metrics	BFN	*Bus Factor Number (BFN)*: is the ratio of core developers present in a project with respect to the total number of developers in a project.	Avelino *et al.* (2016b)
	TFN	Truck Factor Number: is the total number of key developers in a project who can be unexpectedly lost, i.e hit by a truck before the project is discontinued.	Avelino *et al.* (2016b)
	TFC	*Truck Factor Coverage (TFC)*: is the percentage of core developers and their associated authored files in a project.	Avelino *et al.* (2016b)

Our metrics framework calculates the set of metrics from a target open source software project by analyzing its repository through commit information history available in its version control system (GitHub).

Fig. 2.3 presents an overview our metrics framework which consists of two core steps to extract metrics.

Step 1. Mine developers aliases: The author alias mining and consolidation consists of the following substeps (Avelino *et al.* (2016b)) 3]: *(i)* Retrieval all unique dev-emails, where for each git commit has a devemail associated with it, *(ii)* Retrieval of GitHub logins related to developers,*(iii)* Similarity matching of emails and logins: by applying Levenshtein distance (Navarro (2001)), all aliases are compared and, with a certain degree of threshold value, consolidated, and *(iv)* Replacement of author emails by their respective aliases. The final transformation goes through all the commits once again and replacing original authors by their primary alias. As a result, if there is a developer associated with commits with different names, we consider them as a single developer, and the output will be presented in a new aliases list. For example, "Bob.Rob" and "Bob Rob" are different names for a single developer, correspondent to commits, we consider them under the same developer, in a new aliases list, as a single identical substitution. We used a threshold value of 0.8 for the Levenshtein distance to identify different aliases used by the same developer, following the of Joblin et al. (Joblin, Mauerer, Apel, Siegmund & Riehle (2015b)). As future work, we will investigate different similarity techniques and threshold values to assess the accuracy of our approach.

Step 2. Build a social network graph: Social network analysis (SNA) has been used for studying and analyzing the collaboration and organization of developers who are working in teams within software development projects (Meneely & Williams (2011a)). Our developers network model is based on a socio-technical connections during a software project development. Different social network analysis metrics have been devised to describe a community structure and predict quality factors in a software development project. Our approach builds a developer network from the version control system and tracks the change logs. Our adopted developers

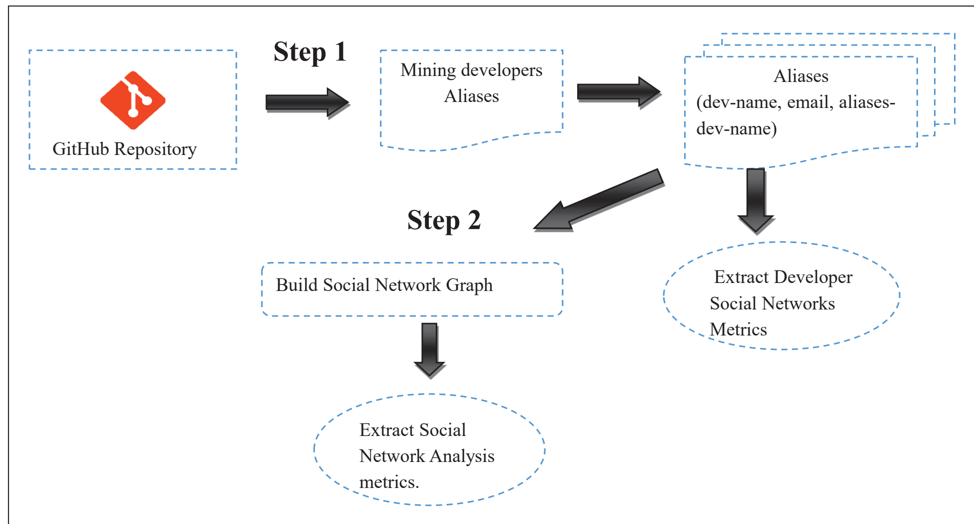


Figure 2.3 An overview of our social-organizational metrics framework.

network is presented as a graph of nodes and edges, where the nodes represent developers and edges are the connections between two developers that are working on the same file and where they make a version control commit within one month of each side. Such social network allows then to calculate the different metrics including the degree centrality, betweenness centrality, closeness centrality, network density, etc. (cf. Table 2.2).

2.3.3 Training model

To prepare our detection model, it is important to perform a correlation analysis within the used metrics. We use our collected dataset to carry out a multi-collinearity analysis between the different metrics using Spearman ρ (Zar (1972b)). Figure 2.4 shows a visualization of the initial auto-correlations among all the variables in the dataset. We observe a few number of correlated metrics (dark-coloured circles in blue or red for positive and negative correlations, respectively). To minimize collinearity among our detector variables, for each pair of metrics having a ρ correlation higher than 0.8, we remove one of the variables. We repeated the technique until there was no pair of metrics that met the criteria. After the correlation analysis, we excluded the following metrics with the highest correlation: NoD, CC, RCD, and ND.

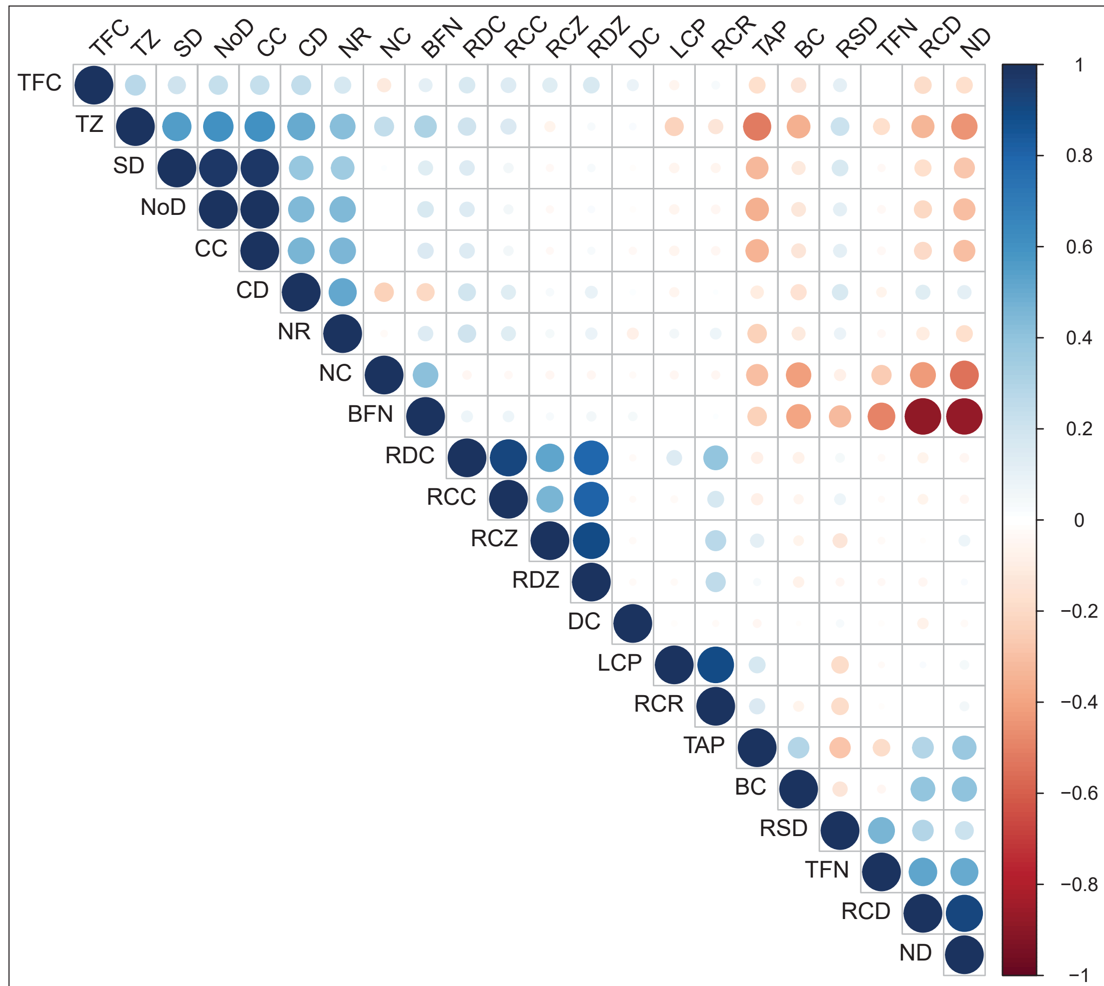


Figure 2.4 Visualization of the metrics correlation analysis.

2.3.4 Data preparation pipeline

As several ML algorithms are available, we considered in our study the adopted C4.5 model, along with five widely used ML models, JRip, Random Forest, Naïve Bayes, and two SVM algorithm instances, SMO, and LibSVM implemented in Weka (Hall *et al.* (2009a)). We first randomly divided our dataset (cf. Section 2.3.1) into training, validation, and test datasets following Hastie et al. guidelines (Hastie, Tibshirani, Friedman & Friedman (2009)). We used the training dataset to fit each individual classifier model, and used the validation dataset to estimate the prediction error for our models comparison and selection as well as hyperparameters tuning, and finally left the test dataset to assess the final chosen models. In our experiments,

we used 50% of the data as training dataset, 25% as validation dataset, and 25% as test dataset (Hastie *et al.* (2009)).

Moreover, as we deal with an imbalanced dataset, we opted for a common resampling technique by applying the Synthetic Minority Oversampling TEchnique (SMOTE) to over-sample minority classes as suggested by Turhan (Turhan (2012)) in building software engineering prediction models. As for the models training and testing, we randomly split our dataset into three parts: training (50%), validation (25%), and test (25%). We use the training dataset to fit each individual classifier model, and the validation dataset to estimate the prediction error/performance for the model's comparison and selection, finally the test dataset to assess the performance of the final studied models.

2.4 Empirical study setup

In this section, we present our empirical study to evaluate our approach. We define four research questions to be addressed and present the experimentation motivation and setup. We then present and discuss the obtained results.

RQ1: To what extent can the employed learner model efficiently detect community smells?

Motivation This first RQ is a sanity check to validate whether our machine learning model could accurately detect the different types of community and does not have a bias toward the detection of any specific smell type.

Approach. To answer RQ1, we perform a comparative study between our used machine learner C4.5 and five widely used models, JRip, Random Forest, Naïve Bayes, and two SVM algorithm instances, SMO, and LibSVM. The data preparation pipeline is explained in Section 2.3.4.

Our models performance comparison is based on widely-used machine learning performance metrics, mean accuracy, FMeasure, and AUC (Area Under The Curve) (Baeza-Yates, Ribeiro-Neto *et al.* (1999); Hastie *et al.* (2009)). Furthermore, we measure for each algorithm its kappa statistic, and the Root Mean Squared Error (RMSE). Moreover, to compare the different classifiers

performance with respect to the eight community smells, we use the non-parametric statistical test Wilcoxon in a pairwise fashion to detect performance differences between the compared algorithms. While the Wilcoxon test verifies the statistical significance of the outcomes, it does not control the difference in magnitude. Hence, we use the non-parametric effect Cliff's delta (δ) (Cliff (1993)). to compute the effect size. The value of effect size is statistically interpreted as:

- Negligible if $\mid d \mid < 0.147$,
- Small if $0.147 \leq \mid d \mid < 0.33$,
- Medium if $0.33 \leq \mid d \mid < 0.474$, or
- High if $\mid d \mid \geq 0.474$.

Moreover, to ensure a fair comparison between the used algorithms, we provide in Table 2.3 the different default parameters used in our comparative study.

Table 2.3 Algorithms parameters configuration.

Algorithm	Parameter	Value
J48	minNumObj	2
	subteeRaising	True
	confidenceFactor	0.25
	unpruned	False
	useLaplace	False
JRip	batchSize	100
	checkErrorRate	True
	numDecimalPlaces	2
Random Forest	maxDepth	0
	numfeatues	0
	numTree	100
	seed	1
Naïve Bayes	usekernelEstimator	False
	useSupervisedDiscretization	False
SMO	buildLogisticsModels	False
	c	1.0
	epsilon	1.0e-12
	filterType	Nomalize training data
	kernel	PolyKernel
	casheSize	250007
	exponent	1.0
	toleranceParamter	0.001
LibSVM	SVMType	C-SVC classification
	doNotReplaceMissingValues	False
	batchSize	100

RQ2: How does the proposed approach perform compared to state-of-the-art community smells detection approaches?

Motivation. While the first research question serves as a sanity check to assess the efficiency of machine learning techniques for the community smells detection problem, RQ2 aims at evaluating the efficiency of the proposed approach compared to recent state-of-the-art techniques for community smells detection to see what improvement can our approach bring.

Approach. We apply our approach, to detect the set of community smells identified from our test dataset of 74 open-source software projects as detailed in Section 2.3.1 using a 10-fold cross validation. We, thereafter, compare our approach with two recent state-of-the-art tools, namely CodeFace4Smells Tamburri *et al.* (2021), and Truck Factor Avelino *et al.* (2016b). CodeFace4Smells is implemented based on a Siemens tool Tamburri *et al.* (2019b) which can detect the four following community smells considered in our experiments: Organizational Silo Effect (OSE), Radio-silence (RS), Prima-donnas Effect (PDE), and Black Cloud Effect (BCE). Truck Factor is another specialized tool that is designed to particularly detect the Track Factor Smell (TFS). Our comparison is based on the detection accuracy. The accuracy is calculated as follows:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \quad (2.1)$$

where TP refers to True Positive, FP for False Positive, FN for False Negative, and TP for True Negative. The values are calculated according the confusion matrix in Table 2.4

Table 2.4 The confusion matrix to calculate the accuracy.

		Actual	
		Positive	Negative
Detected	Positive	TP	FP
	Negative	FN	TN

RQ3. What are the most influential characteristics that can indicate the presence of community smells?

Motivation. The C4.5 classifier results show that we can use a variety of metrics to identify community smells. In addition to the detection accuracy, it is interesting to chart community smells management plans to examine the influence of characteristics of each community smell type (i.e., smells symptoms that are characterized through metrics). Identifying the most influential characteristics of each community smell could be used as an indicator by software project manager to avoid such smells in their organizations. Therefore, in this RQ3, we set out to analyze our model to investigate the characteristics that influence the existence of community smells.

Approach. To answer RQ3, we evaluate the influence and contribution of each metric on the different types of community smells achieved by our C4.5 classifiers. We design an experimental study to estimate variables importance in a random classification (Breiman (2001b); Liaw, Wiener *et al.* (2002)). For each community smell type, we use this technique to compute the Mean Decrease Accuracy (MDA) for each metric. The larger the MDA of the metric is, the more influential the metric to the model is. This experiment is performed using the environment of the importance function of the R randomFoest package. Moreover, we measure the direction of the relationship between each metric and the likelihood of the occurrence of community smell. To do so, we use a Spearman rank correlation (ρ) (Zar (1972b)) to estimate the correlation between each metric and the response (e.g., 1 if a metric indicates the occurrence of a community smell,

and 0 if no smell). A positive correlation in terms of a Spearman rank indicates that the metric has a positive relationship with the likelihood of the occurrence of a community smell, whereas a negative correlation identifies an inverse relationship.

RQ4. Can the proposed metrics improve the performance of community smells detection?

Motivation. Just like code smells (Fowler (2018)), community smells definitions and symptoms are subject to various interpretations, and therefore it can be captured using various metrics. To this end, our aim in extending the list of existing soci-organizational and technical metrics (i.e., metrics tagged as "New" in the last column of Table 2.2) was to cover more salient smells symptoms. Indeed, extending the base of metrics allows to properly characterize community smells properties along with extending the feature space for the classifiers.

Approach. We measure the classification performance, with and without the set of newly introduced metrics (cf. Table 2.2), e.g., Number of active days of an author on a project, Ratio commits per community, Ratio developers per community, Ratio commits per time zone, Ratio developers per time zone, Number of releases in a project and Ratio commits per release. An increase in terms of detection accuracy indicates that the new metrics are useful to better capture the salient characteristics of community smells, whereas a non-increase indicates that the introduced metrics do not extend the classifiers feature space, and thus, they are not relevant to capture smells properties.

RQ5. What is the sensibility of our model with respect to outlier/ influential data points?

Motivation. While we deal with a relatively limited size dataset, we observe from the dataset that the majority of the examined projects come from either the Apache, Eclipse and KDE communities. In particular, we observe that there are clusters of similar projects in the whole dataset, with notable differences among them as can be seen in figure 2.3 and table 2.1 impact on the accuracy and reliability of the obtained results. This suggests the importance of conducting a sensibility analysis to evaluate the possible presence of outliers and/or influential data points that may exert an impact on the classification results. Studying influential instances helps

to identify which training instances should be checked for errors and give an impression on the robustness of the model. Indeed, we might not trust a model if a single instance has a strong influence on the model predictions and parameters (Molnar (2020); Scholbeck, Molnar, Heumann, Bischl & Casalicchio (2020)).

Approach. We study the presence of outliers and/or influential data points. We study the extent to which deleting one of the training instances can affect the resulting model using the deletion diagnostics (Molnar (2020)). A training instance is considered "influential" if its deletion from the training data considerably changes (increases or decreases) the prediction performance of the trained model. The measurement of the influence for all 74 training instances requires to train the model once on the training data and then retrain it n times (i.e., the size of training data) with one of the instances removed each time. We use the influence measure defined by Molnar (Molnar (2020)) for the effect on the model predictions which is defined as follows:

$$influence^{(-i)} = |\hat{y}_j - \hat{y}_j^{(-1)}| \tag{2.2}$$

where $influence^{(-i)}$ returns the differences between prediction performance value of the model with and without the ith instance, over the dataset. The general form of deletion diagnostic measures consists of choosing a performance measure and calculating the difference of the measure for the model trained on all instances and when the instance is deleted. In our experiment, we use the accuracy measure as a proxy of the model performance. We apply the deletion diagnostic of each model of the eight community smell types.

2.5 Results and discussions

In this section, we present and discuss the results of our empirical study with respect to our research questions RQ1–5 set out in Section 2.4.

Results for RQ1: Detection accuracy

Tabale 2.5 reports the obtained results for RQ1 to compare the six considered machine learners C4.5, Random Forest, JRip, SMO, LibSVN and Naïve Bayes in terms of accuracy, Kappa statistic, RMSE and AUC. In more details, figure 2.5 shows the boxplots of each of the eight considered community smells. We observe from both the table and the figure that the C4.5 classifier is the most accurate algorithm achieving an average accuracy score of 96.93% while the LibSVM turns out to be the worst algorithm with a limited average accuracy of 61.65%. A high performance is also achieved by the C4.5 classifier in terms of both performance metrics, Kappa (0.89) and RMSE (0.14). However, in terms of AUC, Random Forest performs better than the other classifier by an average AUC of 0.95 which is slightly better than C4.5 (0.94). The Wilcoxon statistical analysis of the results depicted in the boxplots of Fig. 2.5 indicates that C4.5 is statistically different from the five other classifiers with a high effect size, except the AUC metric when comparing C4.5 and Random Forest. We thus conclude that C4.5 is the best algorithm choice among the compared machine learners.

Table 2.5 The average algorithms performance in terms of Accuracy, Kappa statistic, RMSE, and AUC.

Algorithm	Accuracy	Kappa	RMSE	AUC
J48	**96.93**	**0.89**	**0.14**	0.94
Random Forest	92.22	0.62	0.24	**0.95**
JRip	92.90	0.79	0.24	0.92
SMO	75.67	0.79	0.49	0.8
LibSVM	69.76	0	0.53	0.7
Naïve Bayes	67.56	0.30	0.55	0.5

To get more detailed results, we assess the accuracy detection for each individual community smell type. Table 2.6 reports the precision, recall and F-measure results achieved by the compared algorithms for both classes, i.e., smelly class (S), and non-smelly class (NS). For

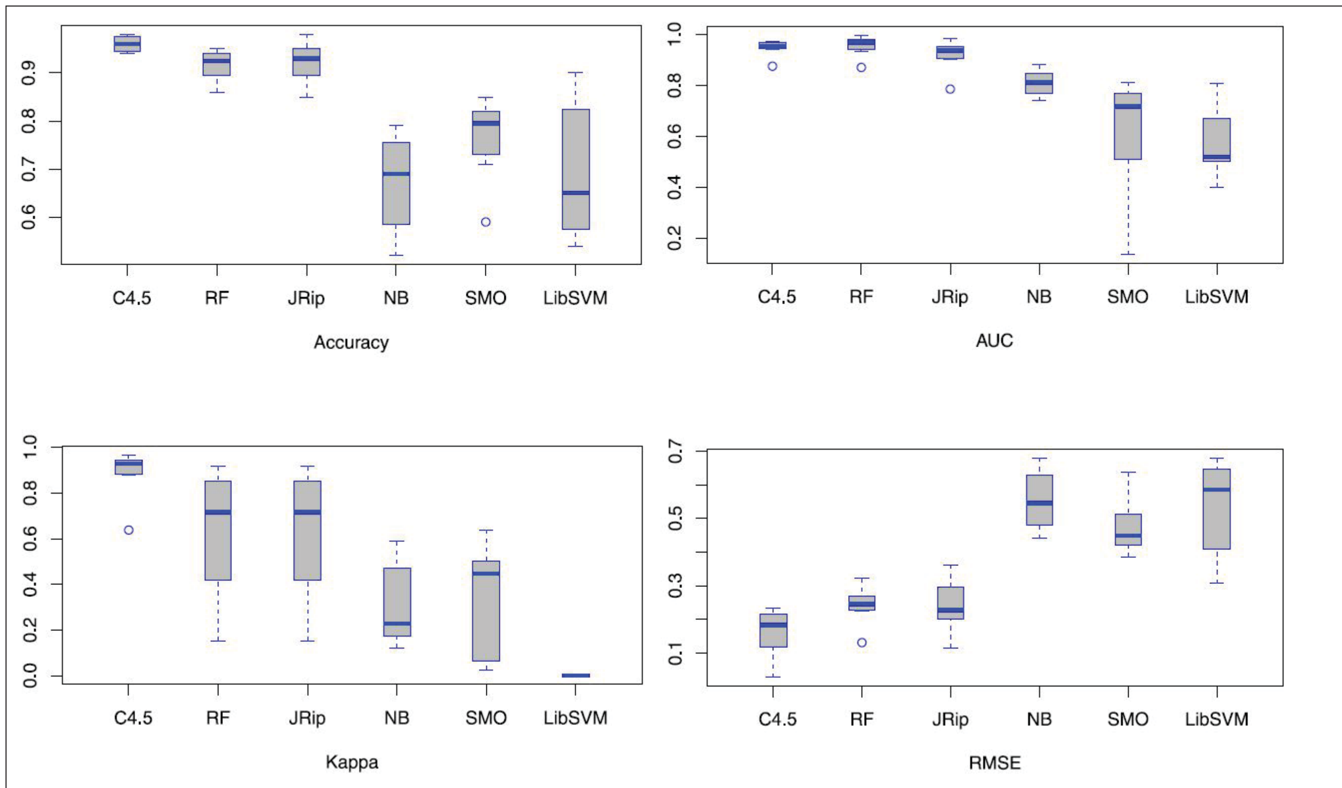


Figure 2.5 Boxplots of the achieved Accuracy, AUC, Kappa, and RMSE results for the eight considered community smells by each of the considered machine learners, C4.5, JRip, LibSVM, Naïve Bayes, Random Forest and SMO.

the smelly class, we observe, from all the considered community smells, that our approach achieved promising detection results with an average precision of 96% and a recall of 91%. The lowest detection results were achieved in the detection of the Sharing Villainy smell (SV), with a precision of 81% and a recall of 59%. This could be due to the low number of smell instances (7 instances) in the base of examples that could be extended to improve the detection performance.

Table 2.6 The achieved results

Classifier	class	OSE			PCE			PDE			SV			OS			SD			RS			TF			Average		
		P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	**P**	**R**	**F**
C4.5	S	0.98	0.96	0.97	0.94	0.96	0.95	1	0.96	0.98	0.81	0.59	0.68	0.94	0.97	0.95	0.97	0.96	0.96	1	0.91	0.95	1	0.94	0.97	**0.96**	**0.91**	**0.93**
	NS	0.88	0.93	0.95	0.96	0.92	0.94	0.98	1	0.99	0.95	0.98	0.97	0.97	0.95	0.96	0.97	0.97	0.97	0.98	1	0.99	0.95	1	0.97	**0.96**	**0.97**	**0.97**
Jrip	S	0.98	0.96	0.97	0.89	0.83	0.86	0.82	0.9	0.86	0.42	0.42	0.42	1	0.91	0.95	1	0.9	0.94	1	0.9	0.94	0.96	0.8	0.87	**0.88**	**0.83**	**0.85**
	NS	0.87	0.93	0.9	0.8	0.87	0.83	0.96	0.92	0.94	0.94	0.94	0.94	0.93	1	0.96	0.93	1	0.96	0.98	1	0.99	0.87	0.97	0.92	**0.91**	**0.95**	**0.93**
LibSVM	S	0.8	1	0.89	0.58	1	0.73	0	0	0	0	0	0.5	0	0	0	0	0	0	0	0	0	0	0	0	**0.17**	**0.25**	**0.27**
	NS	0	0	0	0	0	0	0.72	1	0.84	0.9	1	0.95	0.57	1	0.72	0.6	1	0.75	0.86	1	0.92	0.6	1	0.75	**0.53**	**0.75**	**0.62**
NB	S	0.93	0.47	0.62	0.87	0.69	0.77	0.54	0.61	0.57	0.2	0.71	0.32	0.82	0.7	0.76	0.6	0.2	0.3	0.26	0.8	0.4	0.45	0.9	0.6	**0.58**	**0.64**	**0.54**
	NS	0.29	0.86	0.44	0.68	0.87	0.76	0.84	0.79	0.81	0.96	0.71	0.82	0.77	0.87	0.82	0.62	0.9	0.74	0.95	0.65	0.77	0.8	0.27	0.4	**0.74**	**0.74**	**0.7**
RF	S	0.92	0.96	0.94	0.95	0.9	0.92	0.91	0.9	0.9	0.37	0.16	0.22	0.96	0.91	0.93	0.93	0.96	0.95	1	0.6	0.75	0.91	0.73	0.81	**0.87**	**0.77**	**0.8**
	NS	0.84	0.68	0.75	0.89	0.93	0.91	0.96	0.96	0.96	0.91	0.97	0.94	0.92	0.97	0.95	0.97	0.95	0.96	0.94	1	0.97	0.85	0.95	0.9	**0.91**	**0.93**	**0.92**
SMO	S	0.81	0.96	0.88	0.77	0.73	0.75	0.83	0.47	0.6	0	0	0	0.92	0.67	0.78	0.5	0.23	0.31	0	0	0	0.8	0.53	0.64	**0.58**	**0.45**	**0.5**
	NS	0	0	0	0.68	0.71	0.69	0.82	0.96	0.88	0.9	1	0.95	0.77	0.95	0.85	0.61	0.84	0.71	0.86	0.98	0.92	0.74	0.9	0.81	**0.67**	**0.79**	**0.73**

An interesting aspect that is noticed from our study is related to the nature of our dataset which is highly imbalanced. Indeed, not all community smell types are frequent in real-world systems (cf. 2.7). For example, the Sharing Villainy (SV) smell has the lowest number of smelly instances (7 projects) while 67 projects are not smelly, as it is less frequent compared to other more common smells. On the other side, the Organization Silos Effect (OSE) smell is found in 59 projects while the remaining 15 projects in our datset are non-smelly. Indeed, the class imbalanced datasets occur in many real-world applications and in particular in software engineering, where the class distributions of data are often highly imbalanced. Data resampling is a common approach to solve this problem. We used the Synthetic Minority Over-sampling Technique (SMOTE) (Chawla, Bowyer, Hall & Kegelmeyer (2002)) to deal with such severely imbalanced data. To get more detailed results, we present the precision, recall and F-measure for both classes, i.e., smelly and non-smelly classes, in table 2.6. We observe from the results that C4.5 achieves an average precision and recall score of 96% and 91%, respectively, for the smelly class (S). For the nonsmelly class (NS), the average precision and recall are 96% and 97%, respectively. In particular, for the most imbalanced smell, i.e., SV, C4.5 achieved a relatively acceptable precision and recall of 81% and 59%, respectively for the smelly class (S) even only 7 instances are available. The achieved precision and recall scores are 95% and 98%, respectively, for the non-smelly class (NS). On the other hand, the libSVM and SMO turn out to be the worst classifiers, in particular for the smelly class, as can be shown Table 2.6. Regarding the balance between precision and recall, the best classifier is C4.5 which achieved the highest F-measure in both classes S and NS for the 8 smell types.

Table 2.7 A summary of the identified smells in each studied system.

System	#projects	OSE	BCE	PDE	SV	OS	SD	RS	TF
Apache	38	29	20	7	2	13	21	3	13
Eclipse	7	4	5	1	0	4	2	0	4
KDE	9	9	3	5	3	5	2	4	4
Ganttproject	1	1	0	0	0	0	1	0	1
Qemu/qemu	1	1	1	0	0	1	0	0	0
Nginx/ngin	1	1	0	0	0	0	1	0	0
Bitcoin/bitcoin	1	0	1	1	0	1	0	0	0
Python/cpython	1	0	1	1	0	1	0	0	0
Rails	1	1	1	1	0	1	0	0	0
Audacity	1	1	0	0	0	1	0	1	0
GitLab	1	1	1	1	0	1	0	0	0
Scala	1	0	1	1	0	1	0	0	0
Torando	1	1	1	1	1	1	0	1	1
Arduino	1	1	1	1	0	1	0	0	1
Capistrano	1	1	1	1	1	1	0	1	0
liferay	1	1	1	0	0	0	1	0	1
The practical dev	1	1	1	1	1	0	0	0	1
Floweisshardt/atf	1	1	0	0	0	0	1	0	0
Cloudera	1	1	1	0	0	1	0	0	1
Pdfsam	1	1	0	0	0	0	0	0	0
Squirrel-sql	1	1	0	0	0	0	0	0	1
Direct memory	1	1	1	1	0	0	0	0	1
Flue	1	1	1	0	0	1	1	0	1
Total	74	59	42	21	7	34	30	10	30

To get a more qualitative sense, we present in Table 2.8 six examples from our experiments that are detected by our approach. Looking, for instance, at the Apache/mahout project, we observe that this project experiences two community smells, the organizational silo effect (OSE), and the solution defiance (SD). Hence, the project has been developed in a time window that starts on January 14th, 2008 as a first commit until June 26, 2018 with a life time of 3815 active days when we analyzed it. The Apache mahout project has a total number of 44 developers (NoD) from which 6 are sponsored developers (SD), and 3 communities (NC) and a low graph degree centrality (DC) of 0.008 based on its developers social network. In addition, the total number of time zones is 10 to which developers involved in the project belong to. Indeed, the number of communities in a project indicates the strength and size of the structure for which the different sub-communities in a project and low social network degree centrality may reveal potential disconnections within a community (Tamburri *et al.* (2019b)).

Table 2.8 Examples of projects and related smells.

Project	OSE	BCE	PDE	SV	OS	SD	RS	TFS
Apache/mahut	✔					✔		
Ganttproject	✔					✔		✔
Qemu/qemu	✔	✔			✔			
Bitcoin/bitcoin		✔		✔	✔			
GitLab	✔	✔	✔		✔			
KDE/okular	✔		✔	✔	✔		✔	

On the other hand, the SD smell occurs in this project with different levels of cultural and background within the software development community, and could be related to the high number of different time zones in a project (TZ) which is 10 time zones, and relatively low social

network density (ND) which is equals to 0.257. Indeed, the high TZ metric indicates a high geographical location dispersion within the community, where different cultural backgrounds may contribute to dividing developers into subgroups based on cultural and linguistic differences (Nordio *et al.* (2011)). Hence, different circumstances lead to the occurrence different smells such as organizational silo effect smell (OSE) and a social defiance smell (SD) in the project.

Summary for RQ1. Our machine learning-based approach, csDetector, using C4.5 achieves a high performance for the community smells detection problem with an average accuracy of 96.9%, and an AUC of 0.94.

Results for RQ2: State-of-the-art comparison

Table 2.9 shows the comparison results achieved by csDetector compared to two recent state-of-the-art techniques, Code-Face4smell (Tamburri *et al.* (2021)) and Truck Factor (Avelino *et al.* (2016b)). CodeFace4smell can detect only four out the eight considered community smells namely, OSE, BCE, PDE and RS, while Truck factor is particularly design to detect the TFS smell. We observe from the table that the csDetector outperforms both state-of-the-art techniques in terms of the detection accuracy. For the eight considered smells, csDetector achieves an accuracy in the interval from 94% (for the BCE smell) to 98% (for PDE and RS), while CodeFcae4Smell achieved an accuracy score within the interval 33% (for the OSE smell) to 86% (for the RS smell). Moreover, as shown in table 2.9, although the Truck Factor approach is particularly designed for the detection of the TFS, its accuracy does exceed 84%, while our approach achieves 97% for the truck factor smell.

Table 2.9 The comparison results of csDetector, CodeFace4smell and Truck Factor tools for community smells detection in terms of detection accuracy.

Smells	Detection Accuracy		
	csDetector	CodeFace4Smell	Truck Factor
OSE	95%	33%	-
BCE	94%	75%	-
PDE	98%	75%	-
SV	94%	-	-
OS	95%	-	-
SD	97%	-	-
RS	98%	86%	-
TFS	97%	-	84%

Indeed, one of the limitations of CodeFace4Smells resides in its adopted smells-based detection rules which are based mainly on generic collaboration and communication metrics through a developer social network (based on the change history and mailing lists). Such rules may need a high calibration effort to be adapted to particular contexts. Moreover, its accuracy highly depends on the availability and completeness of the mailing communication traces. On the other hand, our approach learns from real world instances of community smells and uses a wide variety of socio-organizational and technical metrics to capture the key symptoms from different smell types which resulted in an improved detection accuracy.

Summary for RQ2. Our approach, csDetector, outperforms two recent state-of-the-art techniques Code- Face4smell and Truck Factor with a high detection accuracy from 94% to 98% for the eight considered community smells.

Results for RQ3: Influential characteristics

Table 2.10 summarizes all the existing correlations between each community smell type and the list of organizational-social metrics, as well as the average MDA for each metric. To identify the metrics that are highly influential in the detection of a particular community smell, we compute the average MDA for each metric. From table 2.10, we observe that the most influential metrics belong initially to the geographic dispersion metrics dimension and the social network analysis metrics dimension. In particular, the five most influential metrics are the Ratio of commits per time zone (RCZ), the Ratio of developers per time zone (RDZ), the Ratio of developers per community (RDC), the social Graph betweenness centrality (BC), and the social Graph closeness centrality (CC).

Furthermore, we identify the influential characteristics of the analyzed projects and the direction of their relationships with the likelihood of the occurrence of a community smell. The positive correlations are identified by the plus sign (+) while the negative ones are identified with a minus sign (-) in the table. For instance, the NoD metric (the number of developers who modified the code) is identified as an influential metric to the occurrences of the following community smells: Organizational Silo Effect, Prima-donnas Effect, Sharing Villainy, Organizational Skirmish, Radio Silence as it has a positive correlation relationship with these smells and negative ones for others. Looking also at the number of time zones (TZ) metric, we observe that it has a positive correlation with the following smells, Black Cloud Effect (BCE), Sharing Villainy (SV) and Solution Defiance (SD). These findings suggest that more attention should be payed to these particular socio-organizational characteristics within the software project community to avoid such smells.

Table 2.10 The analysis results of the most influential metrics on community smells detection.

Metrics Acronym	OSE	BCE	PDE	SV	OS	SD	RS	TFS	MDA
NOD	+	-	+	+	+	-	+	-	3.31
TAP	-	+	-	+	+	+	-	-	-0.99
LCP	+	-	-	-	+	-	-	-	-0.83
DC	+	-	-	-	-	-	+	-	2.29
BC	-	+	-	+	-	-	-	-	4.23
CC	-	+	+	-	+	-	+	-	4.21
TZ	-	+	+	+	-	+	-	-	2.24
CD	+	-	+	+	-	-	+	-	3.96
RCD	-		+	-	-	-	-	+	1.33
SD	+	-	+	-	-	-	-	-	1.67
RSD	+	-	-	-	-	+	-	-	1.35
ND	-	+	-	+	+	+	+	-	1.09
NR	-	-	+	-	-	-	-	-	0.39
RCR	+	-	-	-	-	-	-	-	2.28
RDC	+	+	+	+	-	+	+	-	4.92
NC	-	+	-	-	-	+	+	-	0.12
RCZ	-	-	+	+	-	-	-	-	8.76
RDZ	+	-	+	-	-	+	+	-	6.23
RCC	-	-	-	-	+	-	-	-	1.59
BFN	+	+	-	+	-	-	+	+	-1.70
FN	-	-	-	-	+	-	-	-	0.50
TFP	+	-	-	-	-	-	+	+	-0.57
TFC	-	+	+	-	-	-	+	+	1.96

Summary for RQ3. Our approach identifies the high influential metrics that can be used as indicators of the existence of community smells. The five most influential metrics are Ratio of commits per time zone, the Ratio of developers per time zone, the Ratio of developers per community, the social Graph betweenness centrality, and the social Graph closeness centrality.

Results for RQ4: Impact of the new metrics

Table 2.11 shows the results of our approach for each community smell, with and without the set of new metrics (cf. Table 2.2), based on the percentage of correctly classified instances (accuracy), kappa statistic, and Root Mean Squared Error (RMSE). We clearly observe from the table that the set of new metrics significantly improves the performance of community smells detection accuracy in all the eight community smell types. For instance, for the OSE smell, the new metrics improved the accuracy from 89% (without the new metrics) to 96% (with the new metrics). Different accuracy improvements were observed also for the PDE, SV, and RS, while the accuracy remains relatively constant for the BCE, OS, SD, and TFS. On average for the eight smell types, the statistical results indicate that the csDetector approach, with the set of new metrics, achieves a high average performance score in terms of accuracy (96.5%), kappa (0.89), and RMSE (0.16) than without the new metrics.

Summary for RQ4. The set of new introduced metrics (cf. Table 2.2) improves the detection performance of four community smell types (OSE, PDE, SV and RS), while maintaining a constant performance in the four remaining smells (BCE, OS, SD and TFS).

Table 2.11 The achieved accuracy, Kappa and RMSE results by csDetector with and without the newly considered metrics.

	MLADCS with new metrics			**MLADCS without new metrics**		
Smell	Accuracy	Kappa	RMSE	Accuracy	Kappa	RMSE
OSE	**95.94%**	**0.877**	**0.201**	89.18%	0.681	0.304
BCE	94.59%	0.889	0.232	94.59%	0.889	0.232
PDE	**98.64%**	**0.966**	**0.116**	93.24%	0.831	0.258
SV	**94.59%**	**0.638**	**0.229**	91.89%	0.457	0.278
OS	95.94%	0.918	0.201	95.94%	0.918	0.201
SD	97.29%	0.943	0.028	97.29%	0.943	0.164
RS	**98.64%**	**0.939**	**0.116**	95.94%	0.833	0.203
TFS	97.29%	0.943	0.164	97.29%	0.943	0.164
Average	**96.61%**	**0.889**	**0.161**	94.42%	0.182	0.225

Results for RQ5: Sensibility analysis

The results of the deletion diagnostics analysis are reported in Fig. 2.6. We train the C4.5 models for each considered smell and check if some training instances were influential overall and for a particular community smell type using the deletion diagnostics. Since the detection of smells is a binary classification problem, we measure the influence as the difference in the model accuracy. An instance is influential if the model accuracy considerably increases or decreases on average in the dataset when the instance is removed from the model training data. The mean value of influence measures for all the eight models over all possible deletions is 1.15%. As can be seen in Fig. 2.6. all the models achieve a median influence score ranging from 0.03% to 2.8%. The most stable models were the track factor (TF), organizational silo effect (OSE) and solution defiance (SD), while other models such as the black cloud effect (BCE) and sharity villainy (SV) exhibited slightly higher sensitivity. We also observe some outlier values in the OSE, PDE, RS and the SD models. Such influence variability could be related to the highly imbalanced dataset

for some smell types such as the SV which have only very few instances. For example, in the SV model, the most influential instance has an influence measure of 4.2% on the accuracy. An influence of 4.2% means that if we remove the instance, the detection accuracy changes by 4.2% on average. This influence is rather low considering that the average accuracy of the SV model is 94.6%. To get a more qualitative sense, we examined this influential instance which is the Apache/Thrift project. Looking in deep into the project's characteristics, we found it with a medium number of developers (293) among all SV smell instances in the dataset, however it has the highest number of time zones (21 different time zones across all developers) making it the project with the lowest number of developers per time zone (16). Indeed such project characteristics make it hard to share knowledge among developers due to a potential lack of synchronized communication. Overall, the generated C4.5 models seem to be stable across all the eight community smells. As part of our future work, we plan to extend the training dataset and study the stability of our models on a larger scale.

Summary for RQ5. The sensibility analysis of our models indicates that the built model is relatively stable with a median sensibility to influential data instances of 1.15%.

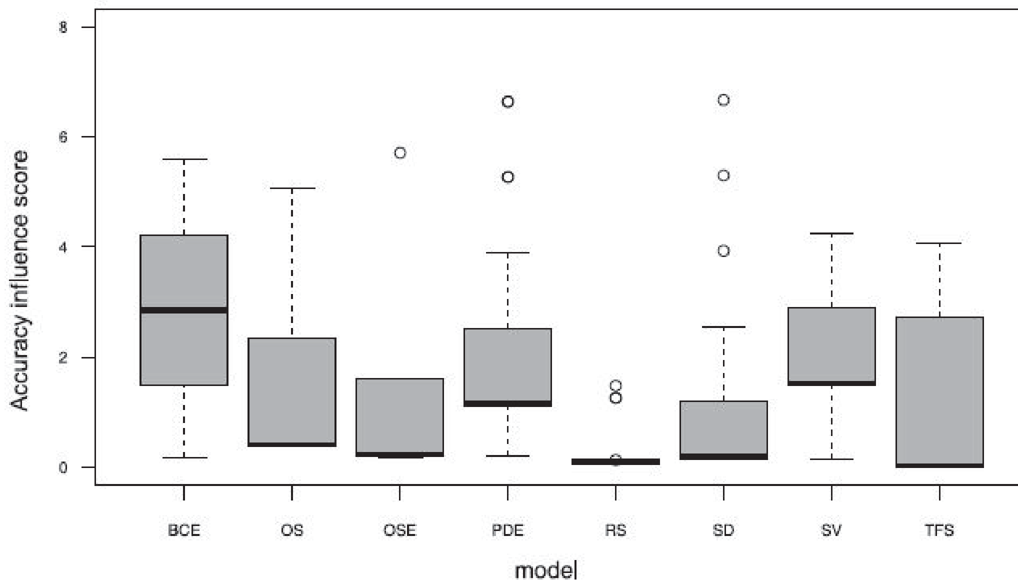


Figure 2.6 Results of the influential instances analysis on the models accuracy.

2.6 Threats to validity

In this section, we discuss the potential threats that might have affected the validity of our results.

2.6.1 Construct validity

Threats to construct validity describe concerns about the relationship between theory and observation and, generally, this type of threats is mainly constituted by imprecisions in performed measurements. Most of what we measured in our approach was based on standard metrics such as precision and recall that are widely accepted as good metrics for the quality of smells detection (Avelino *et al.* (2016b); Fontana, Zanoni, Marino & Mäntylä (2013)). Moreover, we exploited the implementation of prediction models by the Weka framework (Hall *et al.* (2009a)), which is commonly considered as a reliable tool. Another potential threat could be related to the selection of classification techniques. Although we use the C4.5 technique which is known to have high overall accuracy, there are several other classification models that may produce a better classification performance. Hence, to mitigate this threat, we built classifiers using different techniques such as JRip, Random Forest, Naïve Bayes, SMO, and LibSVM. We found that our C4.5 classifiers achieve the highest performance.

The different algorithms used in this study are among the popular and widely applied to recent similar software engineering problems (Tamburri *et al.* (2019c); Avelino *et al.* (2016b); Thongtanunam, Shang & Hassan (2019); Robles & Gonzalez-Barahona (2005)). There could be of course several other decision-tree learning algorithms that could be used in our problem, yet we believe that C4.5 well represents this family of algorithms with high performance. As for overfitting, the optimal solution to challenge our model is through another dataset. But since there is no publicly available set of community smells, we have tested the model using training-validation-test datasets. Our experiments allow the tuning of hyper-parameters with the validation set, and keeps the test set as a truly unseen dataset for assessing the final models performance. One of the issues we had is related to the high data imbalance. Another technical challenge could be related to the amount of data collected to train the machine learning algorithms

which is highly imbalanced. While we used the SMOTE technique, we believe that having a larger dataset would allow more generic and reliable data and best performance. Thus, our future work on community smells will focus on how the problem of imbalanced data can be assessed. Moreover, we plan to assess different other algorithms from different families with different parameter settings to reach higher performance.

Moreover, some threats to internal validity could be related to the social network analysis between committers. Indeed, an inherent characteristics of OSS projects is that they may undergo a radical evolution of their committer base, especially, popular projects with large communities and multi-year history. As a result, some of the committers considered in the social network analysis (SNA) metrics computed from these data could be not active anymore. This inherent issue can exert a powerful influence on the calculated metrics as they may change as the project and the development evolve. As part of our future work, we plan to consider the time factor in our SNA (collaboration recency) and see how community smells may evolve over time.

2.6.2 Internal validity

The internal threat to validity concerns our ability that might have influenced our results, based on the relation between the outcome and the set of organizational-social metrics that are used as independent instances. While we used a collected set of metrics from six different dimensions (i.e., developer social network, social network analysis, community metrics, geographic dispersion, developer network formality, and truck number metrics), adding other new metrics could improve the performance of the classifier. Another important threat to validity could be related to the dataset. The manual identification of community smells is up to the point of a certain degree of error. To mitigate at least potential errors, the authors focused on individually identifying the organizational-social symptoms that characterize the existence of a potential community smell based on established state of the art definitions and guidelines (Tamburri *et al.* (2016); Palomba *et al.* (2018b); Avelino *et al.* (2016b)). For the sake of a clean data, all projects for which there was no total agreement by the three authors was excluded from the dataset. As part of our future work, we plan to validate and extend our dataset in an industrial context. Another

threat to validity could be related to our technique to solve potential committer aliases, based on the Levenshtein distance. This technique may not be robust and may represent some noise in the collected developers identifiers. To better mitigate this issue, we will filter common id strings to increase accuracy, and taking into account the length of e-mail ids as suggested in (Robles & Gonzalez-Barahona (2005)).

2.6.3 External validity

Threats to external validity are connected to the generalization of the obtained results. The influence of the considered metrics on our model is based on the project characteristics. In this study, we investigated the influence of organizational-social metrics that occur as the most influential metrics. Although some projects may not have the same metrics that are highly influential, nevertheless we believe that our results are still of value to determine what kind of organizational-social metrics can be used as an indicator of community smell. Moreover, our approach currently identifies and quantifies eight different community smells and considers them as indicators that may influence social debt existence; on the other hand, there could exist additional community smells not operationalized yet, that may act as critical indicators of the risk of social debt.

Furthermore, rebalancing was not applied to counteract majority class bias that inherently affects our data. Learning from imbalanced data poses new emerging challenges that need to be addressed to build robust models of knowledge from raw data (He & Garcia (2009)). Although the results show that our approach can reasonably learn from both smelly and non-smelly instances for all community smells, replications are needed with larger datasets, using learning techniques specifically designed to deal with skewness in class distribution, to further assess the generality of our models. As part of our future work, we plan to extend our dataset with our industrial partner to add more instances of such smells in order to alleviate possible issues related to such imbalanced datasets.

2.7 Chapter Summary

In this paper, we introduced a community smells detection approach, named csDetector, using C4.5 classifier. We built a learning model based on a set of socio-organizational and technical metrics to detect eight common community smells. We also introduced seven novel metrics to better capture and characterize the key symptoms of community smells. To evaluate our approach, we conducted a set of experiments to assess our adopted C4.5 classifier on eight known community smells to a benchmark of 74 open source projects. Results show that the C4.5 classifier achieves an average AUC of 0.94, suggesting that our classifier can be used to detect community smells. Our experiments indicate also that C4.5 achieves the highest performance compared to five widely used classifiers namely, JRip, Random Forest, Naïve Bayes, SMO and LibSVM. Moreover, our results show that our approach outperforms two recent state-of-the-art community smells detection approaches in terms of accuracy. Our study shows also that the ratio of commits and developers per time zone, the social graph betweenness and closeness centrality are the most influential metrics that indicate the presence of community smells

As future work, we plan to extend our study to other community smell types and projects socio-technical characteristics, while providing ampler empirical evaluation, over multiple open source projects in Github and other software repositories. Moreover, we plan to extend our approach to provide software project manager with community change recommendations to avoid such social debt in their projects. We also plan to assess the impact of community smells on different aspects of software projects.

CHAPTER 3

ON THE DETECTION OF COMMUNITY SMELLS USING GENETIC PROGRAMMING-BASED ENSEMBLE CLASSIFIER CHAIN

Nuri Almarimi[a], Ali Ouni[a], Moataz Chouchen[a], Islem Saidani[a], Mohamed Wiem Mkaouer[b]

[a] Department of Software Engineering and IT, École de Technologie Supérieure
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

[b] Rochester Institute of Technology, 1 Lomb Memorial Dr, Rochester,
NY 14623, United States

Paper published in *ACM/IEEE ICGSE*, September 2020.

Abstract

Community smells are symptoms of organizational and social issues within the software development community that often increase the project costs and impact software quality. Recent studies have identified a variety of community smells and defined them as sub-optimal patterns connected to organizational-social structures in the software development community such as the lack of communication, coordination and collaboration. Recognizing the advantages of the early detection of potential community smells in a software project, we introduce a novel approach that learns from various community organizational and social practices to provide an automated support for detecting community smells. In particular, our approach learns from a set of interleaving organizational-social symptoms that characterize the existence of community smell instances in a software project. We build a multi-label learning model to detect 8 common types of community smells. We use the ensemble classifier chain (ECC) model that transforms multi-label problems into several single-label problems which are solved using genetic programming (GP) to find the optimal detection rules for each smell type. To evaluate the performance of our approach, we conducted an empirical study on a benchmark of 103 open-source projects and 407 community smell instances. The statistical tests of our results show that our approach can detect the eight considered smell types with an average F-measure of 89% achieving a better performance compared to different state-of-the-art techniques. Furthermore,

we found that the most influential factors that best characterize community smells include the social network density and closeness centrality as well as the standard deviation of the number of developers per time zone and per community.

Keywords: Community smells, Social debt, Socio-technical factors, Multi-label learning, Genetic programming, Search-based software engineering.

3.1 Introduction

Modern software engineering is increasingly dependent on the well-being of large globally distributed communities and their social networks in software development. Knowing more about the organizational structures of these communities and their social characteristics as well as the factors that affect their quality is critical to software projects success (Nagappan *et al.* (2008b); Saeki (1995)).

Recent studies explored a set of socio-technical patterns that can negatively impact the organizational health of software projects and coined them as *community smells* (Tamburri *et al.* (2013b, 2015b)). Community smells are connected to circumstances based on poor organizational and social practices that lead to the occurrence of social debt (Tamburri *et al.* (2016, 2015b)). Social debt is connected to negative organized structures that often lead to short and/or long term social issues within a project. These problems could manifest in several forms, *e.g.*, lack of communications, collaboration or coordination among members in a software development community. For example, one of the common community smells, is the "*organizational silo effect*" (OSE) (Palomba *et al.* (2018b); Tamburri *et al.* (2019b)) which manifests as a recurring social network sub-structure featuring highly decoupled developer's community structure. From an analytical perspective, the OSE smell can be interpreted as a set of patterns over a social network graph and could be detectable using different graph connectivity characteristics.

Detecting community smells is still, to some extent, a difficult, time consuming, and manual process. Indeed, there is no consensual way to translate formal definition and symptoms into actionable detection rules. Typically, the number of potential bad organizational practices often

exceeds the resources available to address them. In many cases, mature software projects are forced to be developed with both known and unknown poor socio-technical community practices for lack of resources to deal with every individual community smell. Furthermore, recent studies showed that different types of community smells can have similar symptoms and can thus co-exist in the same project. That is, different symptoms can be used to characterize multiple community smells making their identification even harder and error-prone (Avelino, Passos, Hora & Valente (2016a); Tamburri *et al.* (2019c); Palomba *et al.* (2018b); Tamburri *et al.* (2019b)). For example, the Organizational Silo Effect (OSE) smell is typically associated with the *Solution Defiance* (DF) smell which manifests in the form of independent subgroups in the development team due to the variance in their cultural and experience levels. Although there have been few studies to define, characterize, and identify community smells characteristics/symptoms in software projects, they are applied, in general, to a limited scope, and their generalizability requires a manual effort and human expertise to define and calibrate a set of detection rules to match the symptoms of each community smell type with the actual characteristics of a given software project (Avelino *et al.* (2016a); Tamburri *et al.* (2019c); Palomba *et al.* (2018b); Tamburri *et al.* (2019b)).

In this paper, our aim is to provide an automated technique to detect community smells in software projects. We formulate the problem as a multi-label learning (MLL) problem to deal with the interleaving symptoms of existing community smells by generating multiple smells detection rules that can detect various community smell types. We use the ensemble classifier chain (ECC) technique (Read, Pfahringer, Holmes & Frank (2011)) that converts the detection task of multiple smell types into several binary classification problems for each individual smell type. ECC involves the training of n single-label binary classifiers, where each one is solely responsible for detecting a specific label, *i.e.*, community smell type. These n classifiers are linked in a chain, such that each binary classifier is able to consider the labels identified by the previous ones as additional information at the classification time. For the binary classification, we exploit the effectiveness of genetic programming (GP) (John R. Koza (1992); Glover & Kochenberger (2006); Deb, Pratap, Agarwal & Meyarivan (2002); Ouni, Kessentini,

Inoue & Cinnéide (2017); Kessentini & Ouni (2017); Ouni, Kessentini, Sahraoui & Boukadoum (2013b)) to find the optimal detection rules for each community smell. The goal of GP is to learn detection rules from a set of real-world instances of community smells. In fact, we use GP to translate regularities and symptoms that can be found in real-world community smell examples into detection rules. A detection rule is a combination of socio-technical attributes/symptoms with their appropriate threshold values to detect various types of community smells.

We implemented and evaluated our approach on a benchmark of 103 open source projects hosted in GitHub. We first conducted a survey with developers to validate the identified instances of community smells found in the studied projects. To evaluate the performance of our GP-ECC, the statistical analysis of our results shows that the generated detection rules can identify the eight considered community smell types with an average F-measure of 89% and outperforms state-of-the-art MLL techniques. Moreover, we conducted a deep analysis to investigate the symptoms, *i.e.*, features, that are the best indicators of community smells. We find that standard deviation of the number of developers per time zone and per community, and the social network betweenness, closeness and density centrality within the social network are the most influential characteristics.

To sum up, the paper makes the main following contributions:

- We introduce a GP-based ensemble classifier chain (GP-ECC) approach to detect multiple community smell types that can exist in software projects as a multi-label learning (MLL) problem. To the best of our knowledge, this the first approach that uses MLL and GP for the problem of community smells detection.
- We conduct an empirical study to evaluate our approach on a benchmark of 103 software projects. Our results show that GP-ECC outperforms state-of-the-art single- and multi-label learning techniques.
- We conduct a survey with developers to validate the existence of community smells in our benchmark dat (2020).
- We conduct an exploratory investigation to assess the factors that best characterize community smells in software projects.

Replication package. Our dataset is available online for future extension and replication dat (2020).

Paper organization. Section 3.2 provides the necessary background. In section 3.3, we describe our GP-ECC approach for community smells detection. Section 3.4 presents our empirical evaluation, and discusses the obtained results. Section 3.5 discusses the threats to validity. Finally, in Section 3.6, we conclude and outline our future work.

3.2 Background

3.2.1 Community Smells Definitions

Community smells are defined as a set of social-organizational circumstances that occur within the software development community, having a negative effect on the relations health within the development community which may cause social debt over time Tamburri, Kruchten, Lago & Vliet (2015a). A number of community smells have been defined in the literature. We refer to the following community smell types Tamburri *et al.* (2015a); Avelino *et al.* (2016a):

- **Organizational Silo Effect (OSE):** The OSE smell is manifested when too high decoupling between developers, isolated subgroups, and lack of communication and collaboration between community developers occur. The consequence of this smell is an extra unforeseen cost to a project by wasted resources (*e.g.*, time), as well as duplication of code Tamburri *et al.* (2015a, 2016).
- **Black-cloud Effect (BCE):** The BCE smell occurs when developers have a lack of information due to limited knowledge sharing opportunities (*e.g.*, collaborations, discussions, daily stand-ups, etc.), as well as a lack of expert members in the project that are able to cover the experience or knowledge gap of a community. The BCE may cause a mistrust between members and creates selfish behavioral attitudes Tamburri *et al.* (2015a).
- **Prima-donnas Effect (PDE):** The PDE smell occurs when a team of people is unwilling to respect external changes from other team members due to inefficiently structured collaboration within the community. The presence of this smell may create isolation problems, superiority,

constant disagreement, uncooperativeness and raise selfish team behavior, also called "prima-donnas" Tamburri *et al.* (2015a, 2016).

- **Sharing Villainy (SV):** The SV smell is caused by a lack of high-quality information exchange activities (*e.g.*, face-to-face meetings). The main side effect of this smell limitation is that community members share essential knowledge such as outdated, wrong and unconfirmed information Tamburri *et al.* (2015a).
- **Organizational Skirmish (OS):** The OS smell is caused by a misalignment between different expertise levels and communication channels among development units or individuals involved in the project. The existence of this smell leads often to dropped productivity and affect the project's timeline and cost Tamburri *et al.* (2015a).
- **Solution Defiance (SD):** The solution defiance smell occurs when the development community presents different levels of cultural and experience background, and these variances lead to the division of the community into similar subgroups with completely conflicting opinions concerning technical or socio-technical decisions to be taken. The existence of the SD smell often leads to unexpected project delays and uncooperative behaviors among the developers Tamburri *et al.* (2015a).
- **Radio Silence (RS):** The RS smell occurs when a high formality of regular procedures takes place due to the inefficient structural organization of a community. The RS community smell typically causes changes to be retarded, as well as a valuable time to be lost due to complex and rigid formal procedures. The main effect of this smell is an unexpected massive delay in the decision-making process due to the required formal actions needed Tamburri *et al.* (2015a).
- **Truck Factor Smell (TFS):** It occurs when most of the project information and knowledge are concentrated in one or few developers. The presence of this smell eventually leads to a significant knowledge loss due to the turnover of developers Ferreira, Avelino, Valente & Ferreira (2016); Avelino *et al.* (2016a).

In this paper, we focus primarily on these smells as they are widely studied and most occurring in the software industry as well as in open-source projects based on recent studies Tamburri *et al.* (2016, 2015a); Palomba *et al.* (2018b); Ferreira *et al.* (2016).

3.2.2 Search Based Software Engineering

Search-Based Software Engineering (SBSE) consists of the application of a computational search to solve optimization problems in software engineering Harman & Jones (2001). The term SBSE was coined by Harman and Jones in 2001, and the goal of the field is to move software engineering problems from human-based search to machine-based search, using a variety of techniques from the metaheuristic search and evolutionary computation paradigms Harman & Jones (2001); Harman (2007).

SBSE provides best practice in formulating a software engineering problem as a search problem, by defining a suitable solution representation, fitness function, and solution change operators. Indeed, there are a multitude of search algorithms ranging from single to many-objective techniques that can be applied to solve that problem Harman *et al.* (2012b); Ouni *et al.* (2016a,b, 2013a,c); Ouni *et al.* (2017); Ouni, Kessentini, Sahraoui & Hamdi (2012).

In this paper, we apply SBSE to the problem of community smells detection in software projects. Hence, we show how genetic programming (GP) can effectively explore a large space of solutions, and provide intelligible detection rules with ECC.

3.2.3 Multi-label learning

Multi-label learning (MLL) is the machine learning task of automatically assigning an object into multiple categories based on its characteristics de Carvalho & Freitas (2009); Tsoumakas, Katakis & Vlahavas (2010a); Tsoumakas & Katakis (2007); Read *et al.* (2011). Single-label learning is limited by one instance with only one label. MLL is a non-trivial generalization by removing the restriction and it has been a hot topic in machine learning de Carvalho & Freitas (2009); Tsoumakas *et al.* (2010a). MLL has been explored in many areas in machine learning

and data mining fields through classification techniques Tsoumakas & Katakis (2007). There exists different MLL techniques including (1) problem transformation methods and algorithms, *e.g.*, the classifier chain (CC) algorithm Read *et al.* (2011), the binary relevance (BR) algorithm Tsoumakas & Katakis (2007), label powerset (LP) algorithm Tsoumakas, Katakis & Vlahavas (2010b), and (2) algorithm adaptation methods such as the K-Nearest Neighbors (ML.KNN) Zhang & Zhou (2007), as well as (3) ensemble methods such as the ensemble classifier chain (ECC) Read *et al.* (2011) and random k-labelset (RAKEL) Tsoumakas & Katakis (2007). MLL was successfully applied to solve different software engineering problems McIlroy, Ali, Khalid & Hassan (2016); Podgurski *et al.* (2003); Feng & Chen (2012); Xia, Feng, Lo, Chen & Wang (2014).

The Classifier Chain (CC) model. The CC model combines the computational efficiency of the BR method while still being able to take the label dependencies into account for classification. With BR, the classifier chains method involves the training of q single-label binary classifiers and each one will be solely responsible for classifying a specific label $l_1, l_2, ..., l_q$. The difference is that, in CC, these q classifiers are linked in a chain $\{h_1 \rightarrow h_2 \rightarrow \ ... \rightarrow h_q\}$ through the feature space. That is, during the learning time, each binary classifier h_j incorporates the labels predicted by the previous $h_1, ..., h_{j-1}$ classifiers as additional information. This is accomplished using a simple trick: in the training phase, the feature vector x for each classifier h_j is extended with the binary values of the labels $l_1, ..., l_{j-1}$.

The Ensemble Classifier Chain (ECC) model. One of the limitation of the CC model is that the order of the labels is random. This may lead to a single standalone CC model be poorly ordered. Moreover, there is the possible effect of error propagation along the chain at classification time, when one (or more) of the first classifiers predict poorly Read *et al.* (2011). Using an ensemble of chains, each with a random label order, greatly reduces the risk of these events having an overall negative effect on classification accuracy. A majority voting method is used to select the best model. Moreover, a common advantage of ensembles is their well-known effect of generally increasing overall predictive performance Read *et al.* (2011).

In our study, we bridge the gap between MLL and SBSE based on the ECC method to solve the problem of community smells detection, where each project may contain different interleaving community smells, *e.g.*, OSE SV and BCE. For the binary labels, our ECC model adopts a search-based approach using genetic programming (GP) to learn detection rules for each smell type.

3.3 Approach

In this section, we provide the problem formulation for community smells detection as a MLL problem. Then, we describe our approach.

3.3.1 Problem formulation

We define the community smells detection problem as a multi-label learning problem. Each community smell type is denoted by a label l_i. A MLL problem can be formulated as follows. Let $X = R^d$ denote the input feature space. $L = \{l_1, l_2, ...l_q\}$ denote the finite set of q possible labels, *i.e.*, smell types. Given a multi-label training set $D = \{(x_1, y_1), (x_2, y_2), ...(x_N, y_N)\}(x_i \in X, y_i \subseteq L)$, the goal of the multi-label learning system is to learn a function $h : X \rightarrow 2^L$ from D which predicts a set of labels for each unseen instance based on a set of known data.

In our approach, we used the ensemble classifier chain (ECC) model Read *et al.* (2011). While the existing MLL methods, *e.g.*, BR and LP are flexible for solving MLL problems, they have the limitation of ignoring the correlations between labels. To address this issue in our approach, we adopt the ECC model Read *et al.* (2011), an extension of BR to exploit the advantage of label correlations, *i.e.*, smells correlation.

3.3.2 Approach Overview

Our approach starts from the observation that it is easier for developers to identify a set of detection rules to match the symptoms of a community smell with the actual characteristics of a given software project rather than relying on manual effort and human expertise Tamburri

et al. (2013b). The main goal of our approach is to generate a set of detection rules for each community smell type while taking into consideration the dependencies between the different smell types and their interleaving symptoms.

Figure 3.1 presents an overview of our approach to generate community smells detection rules using the GP-ECC model. Our approach consists of two phases: training phase and detection phase. In the training phase, our goal is to build an ensemble classifier chain (ECC) model learned from real-world community smells identified from software projects based on several GP models for each individual smell. In the detection phase, we apply this model to detect the proper set of labels (*i.e.*, types of community smells) for a new unlabeled data (*i.e.*, a new project).

Our framework takes as inputs a set of software projects with known labels, *i.e.*, community smells (phase A). Then, extracts a set of features characterizing the considered community smell types from which a GP algorithm will learn (phase B). Next, an ECC algorithm will be built (phase C). The ECC algorithm consists of a set of classifier chain models (CC), each with a random label order. Each CC model, learns eight individual GP models for each of the eight considered smell types. The ith binary GP detector will learn from the training data while considering the existing i already detected smells by the $i-1$ detected smells generate the optimal detection rule that can detect the current i^{th} smell. In total, the ECC trains n multi-label CC classifiers $CC_1, ..., CC_n$; each classifier is given a random chain ordering; each CC builds 8 binary GP models for each smell type. Each binary model uses the previously predicted binary labels into its feature space. Then, our framework searches for the near optimal GP-ECC model from these n multi-label chain classifiers using an ensemble majority voting schema based on each label confidence Read *et al.* (2011). In the detection phase, the returned GP-ECC model is a machine learning classifier that assigns multiple labels, *i.e.*, community smells types, to a new project based on its current features, *i.e.*, its socio-technical characteristics (phase D). In the next subsections, we provide the details of each phase.

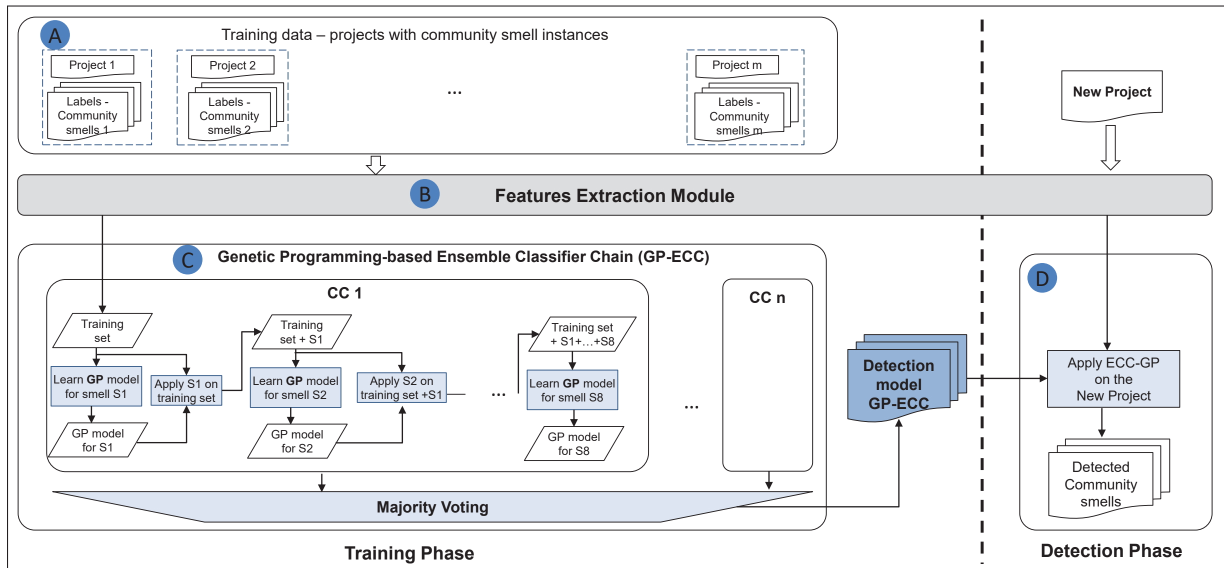


Figure 3.1 The community smells detection framework using the GP-ECC model.

3.3.3 Phase A : Training data collection

An important step to solving the problem of community smells detection is to prepare the learning dataset.

Projects selection: To build a base of real-world community smell examples that occur in software projects, we selected a set of software projects which are diverse in nature (*e.g.*, size, application domains, etc.) that have experienced community smells. We considered a set of open-source projects from Github to access to their development history. The considered filters to collect the training data are the following:

- *Commit size*: the projects vary from medium size >10 KLOC, large size (10- 60) to very large size < 60 KLOC.
- *Community size*: the projects team size vary from medium (<100 members), large (100-900), to very large (> 900).
- *Programming language*: the selected projects are implemented in different programming languages, including Java, C#, Python and C.
- *Existence of community smells:* the project should contain at least one community smell.

Base of examples: One of the main aspects of any attempt to apply a machine learning approach is to collect a base of examples to be used to train the model. We collected a set of community smell instances based on the symptoms and guidelines provided in the literature Tamburri *et al.* (2019); Tamburri *et al.* (2016, 2015a); Palomba *et al.* (2018b); Tamburri *et al.* (2018); Tamburri *et al.* (2019, 2013); Ferreira *et al.* (2016). We manually analyzed the selected projects to identify potential smell occurrences.

Table 3.1 Questionnaire filled in by the study participants.

Part 1: Background Information:	
1	What is your project name on GitHub ? * [*Required Answer*]
2	How do you describe your occupation in this project? [*Student − Part-time employee − Full-time employee − Unemployed − Retired − Other*]
3	How long time you have been working in open-source projects? [*Less than 3 years − 3 years or more − Other*]
Part 2: Social aspects perception	
4	Do you think there is a lack of communication and cooperation within the development team (share knowledge, experiences, exchange activities, etc.) in your project?
5	Do you think there is a delay in communications between developers in your project?
6	Do you think there is/are developer(s) in your project working on the same code and communicating by means of a third person?
7	Do you think there is/are in your project developer(s) who have selfish and/or egoistical behaviors?
8	Do you think there are different communication and expertise levels that may cause misalignment within the development team in your project?
9	Do you think there are different levels of cultural and experiential backgrounds in the development team?
10	Do you think there is/are isolated subgroup(s) within the development team of your project?
11	Do you think there is in your project a formal and complex organizational structure (with potentially unnecessary "regular procedures")?
12	Do you think there is in your project some unique knowledge and information brokers toward different developers?
13	Do you think there is in your project a risk that a core developer who can unexpectedly leave the project and can influence the development process?
14	Do you think there is in your project a waste of resources (*e.g.*, time) over the development life-cycle?

To collect our base of examples, the authors checked individually all identified community smells if they match with the state-of-the-art definitions, characteristics and symptoms. All community smell instances that did not reach a full agreement were excluded from our base of examples. After the manual inspection of potential existence of community smells, we finally ended with 103 projects that have diverse types of community smells dat (2020).

As an attempt to validate our identified smells, we conducted a survey with the original developers of the selected projects to get their feedback by following an opt-in strategy Hunt, Shlomo & Addington-Hall (2013) for our survey. While the survey will help us to validate our identified symptoms, it can also help to understand whether developers are conscious of the presence of such smells in their projects. We extracted from GitHub the email address of the developers who have acted in a project at least 30 commit changes during the last 12 months

and participated in the project in the last 3 years. In such a way, we focused only on developers having adequate experience with the project's community Sugar (2014). Before we sent the survey's questions, we first sent an email asking permission to participate in our study. As a result, we obtained a positive response from 62 out of 432 developers (14%) who were later contacted with the actual survey. We received answers from developers of 31 different projects in the dataset, which covered 29.3% of all considered smells. In our study, we are aware that the different opinions on this survey may not be necessarily generalized, but this analysis helps us to confirm whether our smells symptoms analysis match with the participants perception on such smells in their projects.

The survey's link was sent via email to all participants with a brief introduction. The list of questions is divided into two main parts as shown in Table 4.1. The first part consists of three control questions on the project name and background information about the participants occupation and experience with the project.

Then, we asked developers to rate the validity of 11 statements that we extracted from the community smells definitions (cf. Section 3.2) using a 5 point Likert scale Likert (1932) ranging between "*Strongly Disagree*" to "*Strongly Agree*". This part presents typical situations in which community smells occur without mentioning the term social debt or smells in the questionnaire. To avoid any possible bias in the responses, we did not ask the developers directly to rate how healthy was their community. For instance, the statement "*Do you think there is a lack of communications and cooperation within the development team (share knowledge, experiences, exchange activities, etc.) in your project?*" was aimed at understanding whether developers actually recognize the presence of the symptoms of an OSD smell in their project. The complete questions list of our survey presented in Table 4.1. After collecting the participants answers, we compared their answers with the smell instances detected manually in our base of examples to make sure that they match.

3.3.4 Phase B : Features Extraction Module

To capture community smells symptoms, we rely on a set of metrics defined in previous studies Avelino *et al.* (2016a); Damian A. Tamburri,Simone Gatti (2016); Tamburri *et al.* (2019); Nagappan, Murphy & Basili (2008a); Pinzger, Nagappan & Murphy (2008a); Nordio *et al.* (2011), as well as a new set of generated metrics to capture more community-related proprieties that can be mined from the projects history. These metrics analyze different aspects in software development communities including organizational dimensional, social network characteristics, developers collaborations, and truck numbers. Table 2.2 depicts our list of considered features. Our proposed metrics extend existing metrics to provide more details including developers social network, community structures, geographic dispersion, and developer network formality. For instance, the geographic dispersion metrics, *e.g.*, the *average number of commits per time zone* and the *standard deviation of developers per time zone* would capture the distribution of commits and developers per time zone. Our features extraction module calculates the set of metrics from a given project by analyzing its repository through commit information history available in its version control system, *e.g.*, GitHub. Figure 3.2 depicts an overview of our features extraction module which consists of two main steps to extract metrics.

Step 1. Mine developers aliases: The author alias mining and consolidation consists of the following sub-steps Avelino *et al.* (2016a): (*i*) retrieval all unique dev-emails, where for each git commit has a dev-email associated with it, (*ii*) Retrieval of GitHub logins related to developers, (*iii*) similarity matching of emails and logins: by applying Levenshtein distance Avelino *et al.* (2016a), all aliases are compared and, with a certain degree of threshold value at most one, consolidated, and (*iv*) replacement of author emails by their respective aliases Avelino *et al.* (2016a). The final transformation goes through all the commits once again and replacing original authors by their main alias. As a result, if there is a developer associated with commits with different names, we consider them as a single developer, and the output will be presented in a new aliases list. For example, “Bob.Rob" and “Bob Rob" are different names for a single developer associated to commits. Thus, we consider them as the same developer in a new aliases list as a single identical substitution.

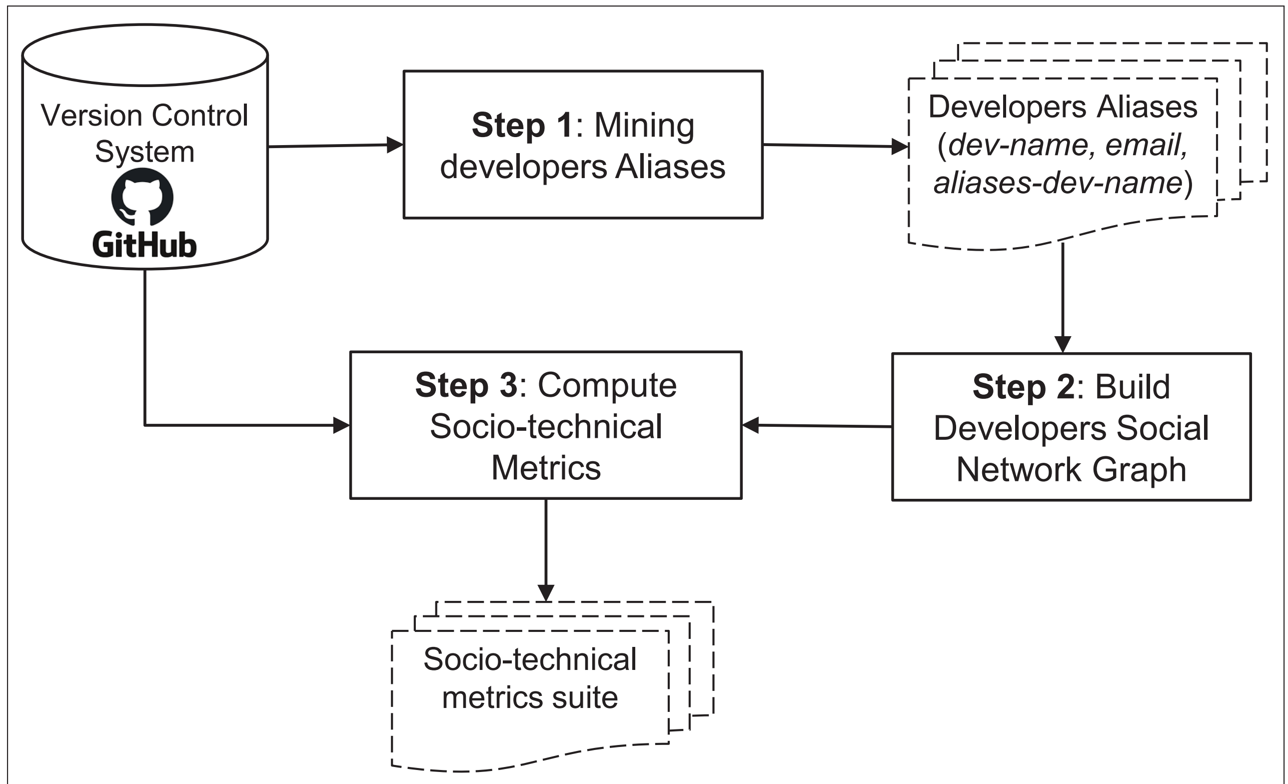


Figure 3.2 An overview of the features extraction module.

Step 2. Build a social network graph: Social networks analysis (SNA) have been used for studying and analyzing the collaboration and organization of developers who are working in teams within software development projects Meneely & Williams (2011b). Our developers network model is based on a socio-technical connection during a software project development. Different social network analysis metrics have been devised to describe a community structure and predict quality factors in a software development project. Our approach builds a developer network from the version control system by tracking the change logs. Our adopted developers network is presented as a graph of nodes and edges, where the nodes represent developers and edges are the connections between two developers that are working on the same file and where they make a version control commit within one month of each side. Such social network allows then to calculate the different metrics including the degree centrality, closeness centrality, network density, etc. (cf. Table 3.2).

Step 3. Compute Socio-technical Metrics: In this step, we use the collected informations and social network graph to compute a variety of 30 socio-technical metrics as described in Table 3.2

Table 3.2 Socio-technical metrics framework.

Dimension	ID	Definition	Ref.
Developer Contributions metrics	NoD	*Number of developers (NoD)*: the total number of developers who have changed the code in a project.	Nagappan *et al.* (2008a)
	NAD	*Number of Active Days of an author on a project (NAD)* : the percentage of the total number of active days for each developer with respect to a project's lifetime on the total number of developers in a project.	Nagappan *et al.* (2008a)
	NCD	*Number of Commits per Developer in a project (NCD)* : the total number of times that the code has been changed by a developer with respect to the total number of commits and the total number of developers in a project.	Nagappan *et al.* (2008a)
	SDC	*Standard Deviation of Commits per developer in a project (SDC)* : the standard definition of commits per developer in a project. It provides a view of the distribution of the developers contributions.	Nagappan *et al.* (2008a)
	NCD	*Number of Core Developers (NCD)*: the total number of core developers in a project. A developer is considered as a core community member if he/she has a larger degree than peripheral developers within the developer's social network.	Tamburri *et al.* (2019)
	PCD	*Percentage of Core Developers (PCD)*: the percentage of core developers with respect to the total number of developers.	Tamburri *et al.* (2019)
	NSD	*Number of Sponsored Developers (NSD)*: the total number of sponsored developers in a project. We consider a developer that hold a sponsored status if at least 90% of her/his commits are performed during weekdays and the working day time (8am-6pm).	Tamburri *et al.* (2019)
	PSD	*Percentage of Sponsored Developers (PSD)*: the percentage of sponsored developers over the total number of developers.	Tamburri *et al.* (2019)
Social Network Analysis metrics	GDC	*Graph Degree Centrality (GDC)*: the number of connections that a developer has. The more connections with others a developer has, the more important the developer is.	Pinzger *et al.* (2008a)
	SDD	*Standard Deviation of a graph Degree centrality in a project (SDD)* : the standard deviation of the degree centrality (DC) of each developer in a project. It provides a view of the distribution of DC in a project.	Tamburri *et al.* (2019)
	GBC	*Graph Betweenness Centrality (GBC)*: a measure of the information flow from one developer to another and devised as a general measure of social network centrality. It represents the degree to which developers stand between each other. A developer with higher BC would have more control over the community as more information will pass through her/him.	Pinzger *et al.* (2008a)
	GCC	*Graph Closeness Centrality (GCC)*: a measure of the distance between a developer to other developers in the network. This metric is strongly influenced by the degree of connectivity of a network.	Pinzger *et al.* (2008a)
	ND	*Network Density (ND)*: a measure of a social network as a dense or sparse graph.	Tamburri *et al.* (2019)
Community metrics	NC	*Number of Communities (NC)*: the total number of communities in a project.	Tamburri *et al.* (2019)
	ACC	*Average of Commits per Community (ACC)*: the average number of commits per community in a project.	New
	SCC	*Standard deviation of Commits per Community (SCC)*: the standard deviation of commits performed by each community with respect to the total number of commits in a project.	New
	ADC	*Average number of Developers per Community (ADC)*: the average number of developers per community in a project with respect to the total number of developers in a project.	New
	SDC	*Standard deviation of Developers per Community (SDC)*: the standard deviation of commits performed by each community with respect to the total number of commits in a project.	New
Geographic Dispersion metrics	TZ	*Number of Time Zones (TZ)*: the total number of different time zones of developers in a project.	Nordio *et al.* (2011)
	ACZ	*Average of Commits per time Zone (ACZ)*: the average number of commits per time zone in a project.	New
	SCZ	*Standard deviation of Commits per time Zones (SCZ)*: the standard deviation of commits performed in each time zone with respect to the total number of commits in a project.	New
	ADZ	*Average number of Developers per time Zone (ADZ)*: the average number of developers per time zone in a project.	New
	SDZ	*Standard deviation of Developers per time Zones (SDZ)*: the standard deviation of developers per time zones in a project.	New
Formality metrics	NR	*Number of Releases in a project (NR)*: the total number of releases delivered in a project.	A. Gopal & S. Krishnan (2002)
	PCR	*Parentage of Commits per Release (PCR)* : the percentage of commits of each release over the total number of releases in a project.	New
	SCR	*Standard deviation of Commits per Release (SCR)* : the standard deviation of developers per release in a project.	New
	FN	*Formal Network (FN)*: the number of milestones assigned to the project with respect to the lifetime of the project.	Damian A. Tamburri (2016)
Truck Number metrics	BFN	*Bus Factor Number (BFN)*: is the percentage of active developers present in a project with respect to the total number of developers.	Cosentino, Izquierdo & Cabot (2015)
	TFN	*Truck Factor Number (TFN)*: the number of key developers in a project who can be unexpectedly lost, *i.e.*, hit by a truck before the project is discontinued.	Avelino *et al.* (2016a)
	TFC	*Truck Factor Coverage (TFC)*: the percentage of core developers and their associated authored files in a project.	Avelino *et al.* (2016a)

3.3.5 Phase C : Genetic Programming-based Ensemble Classifier Chain (GP-ECC)

As explained earlier in Sections 3.3.2 and 4.2.3, our approach is based on the ECC method Read *et al.* (2011) that transforms the multi-label learning task into multiple single-label learning tasks. Our multi-label ECC model aims at building a detection model to detect different instances of community smells in a software project. Each classifier chain (CC) builds a GP model for each smell type while considering the previously detected smells (if any), *i.e.*, each binary GP model uses the previously predicted binary labels into its feature space. Our choice for GP is motivated by the high performance of GP in solving challenging software engineering problems including design defects, code smells and anti-patterns detection Kessentini & Ouni (2017); Ouni *et al.* (2017); Ouni *et al.* (2013b); Ouni, Gaikovina Kula, Kessentini & Inoue (2015).

Algorithm 3.1 High level pseudo code of the adopted MOGP

1: Create an initial population P_0
2: Create an offspring population Q_0
3: $t = 0$
4: **while** stopping criteria not reached **do**
5: $R_t = P_t \cup Q_t$
6: F = fast-non-dominated-sort(R_t)
7: $P_{t+1} = \emptyset \; and \; i = 1$
8: **while** $| P_{t+1} | + | F_i | \leqslant N$ **do**
9: Apply crowding-distance-assignment(F_i)
10: $P_{t+1} = P_{t+1} \cup F_i$
11: $i = i + 1$
12: **end while**
13: $Sort(F_i, \prec n)$
14: $P_{t+1} = P_{t+1} \cup F_i[N- | P_{t+1} |]$
15: Q_{t+1} = create-new-pop(P_{t+1})
16: t = t+1
17: **end while**=0

In our approach, we adopted the Multi-objective Genetic Programming (MOGP) as search algorithm to generate smells detection rules. MOGP is a powerful and widely-used evolutionary algorithm which extends the generic model of GP learning to the space of programs. Unlike other evolutionary search algorithms, in MOGP, solutions are themselves programs following a

tree-like representation instead of fixed length linear string formed from a limited alphabet of symbols John R. Koza (1992).

As described in Algorithm 3.1, MOGP starts by randomly creating an initial population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_0 of size N (line 5). *Fast-non-dominated-sort* Deb *et al.* (2002) is the technique used by MOGP to classify individual solutions into different dominance levels (line 6) Deb *et al.* (2002). The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0. Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When MOGP has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance Deb *et al.* (2002) to make the selection (line 9). This parameter is used to promote diversity within the population. The front F_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of F_i are chosen (line 14). Then, a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4).

We describe in the following subsections the three main adaptation steps: (*i*) solution representation, (*ii*) the generation of the initial generation (*iii*) fitness function, and (*iv*) change operators.

(*i*) Solution representation. A solution consists of a rule that can detect a specific type of community smells in the form of IF-THEN:

In MOGP, a solution is represented as a tree composed of terminals and functions. The terminals correspond to different socio-technical specific features (cf. Table 3.2) with their threshold

values. The functions that can be used between these metrics are logic operators OR (union), AND (intersection), or XOR (eXclusive OR). A solution is represented as a tree a binary tree such that: each leafnode (Terminal) contains one of metrics described in Table 3.2 and their corresponding threshold values generated randomly. Each internal-node (Functions) belongs to the Connective (logic operators) set $C = \{AND, OR, XOR\}$. The threshold values are selected randomly along with the comparison and logic operators. Figure 4.4 shows a simplified example of a solution for the OSE smell using the metrics GDC, ADC, GBC and SDZ.

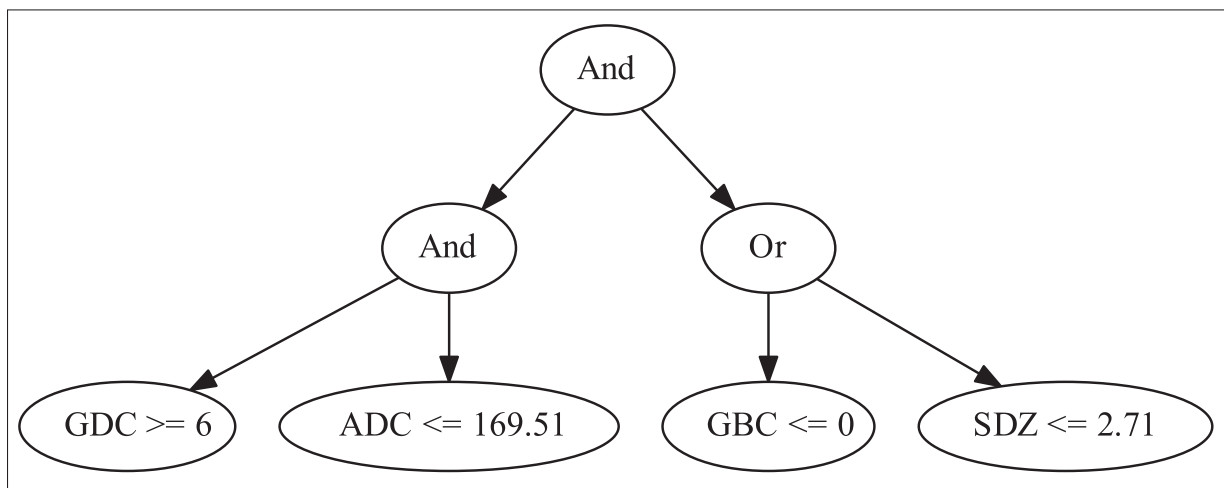


Figure 3.3 A simplified example of a solution for OSE smell.

(*ii*) Generation of the initial population. The initial population of solutions is generated randomly by assigning a variety of metrics and their thresholds to the set of different nodes of the tree. The size of a solution, *i.e.*, the tree's length, is randomly chosen between lower and upper bound values. These two bounds have determined and called the problem of bloat control in GP, where the goal is to identify the tree size limits. Thus, we applied several trial and error experiments using the HyperVolume (HP) performance indicator Angeline (1994) to determine the upper bound after which, the sign remains invariant.

(*iii*) Fitness function. The fitness function evaluates how good is a candidate solution in detecting community smells. Thus, to evaluate the fitness of each solution, we use two objective functions, based on two well-known metrics Kessentini & Ouni (2017); Ouni *et al.* (2017); Harman & Clark (2004), to be optimized, *i.e.*, precision and recall. The precision objective

function aims at maximizing the detection of correct community smells over the list of detected ones. The recall objective function aims at maximizing the coverage of expected community smells from the base of examples over the actual list of detected smells. Precision and recall of a solution S are defined as follows.

$$Precision(S) = \frac{|\{\text{Detected smells}\} \cap \{\text{Expected smells}\}|}{|\{\text{Detected smells}\}|} \tag{3.1}$$

$$Recall(S) = \frac{|\{\text{Detected smells}\} \cap \{\text{Expected smells}\}|}{|\{\text{Expected smells}\}|} \tag{3.2}$$

(*iv*) Change operators. Crossover and mutation are used as change operators to evolve candidate solutions towards optimality.

Crossover. We adopt the “standard" random, single-point crossover. It selects two parent solutions at random, then picks a sub-tree on each one. Then, the crossover operator swaps the nodes and their relative subtrees from one parent to the other. Each child thus combines information from both parents.

Mutation: It can be applied either to a function node or a terminal node. This operator can modify one or many nodes. For a selected solution, the mutation operator first randomly selects a node in the tree. Then, if the selected node is a terminal (metric), it is replaced by another terminal (metric or another threshold value); if the selected node is a function (AND-OR-XOR operators), it is replaced by a new function (*e.g.*, AND becomes OR). If a tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree.

ECC majority voting. As shown in Figure 3.1, for each CC, MOGP will generate an optimal rule for each community smell type, *i.e.*, binary detection. Then, ECC allows to find the best CC that provides the best MLL from all the trained binary models. Each CC_i model is likely to be unique and able to achieve different multi-label classifications. These classifications are summed by label so that each label receives a number of votes. A threshold is used to select the most popular labels which form the final predicted multi-label set. This is a generic voting

scheme and it is straightforward to apply an ensemble of any MLL transformation method Read *et al.* (2011).

3.3.6 Phase D : Detection Phase

After the GP-ECC model is constructed, in the training phase, it will be then used to detect a set of labels for a new project. It takes as input the set of features extracted from a given project using the feature extraction module. As output, it returns the detection results for each individual label, *i.e.*, community smell type.

3.4 Evaluation

This section reports our empirical study to evaluate our approach including the research questions, experiments setup and results.

3.4.1 Research Questions

We designed our empirical study to answer the three following research questions.

- **RQ1: (Performance)** How accurately can our GP-ECC detect community smells?
- **RQ2: (Sensitivity)** What types of community smells does our GP-ECC approach detect correctly?
- **RQ3: (Features influence)** What are the most influential features that can indicate the presence of community smells?

3.4.2 Analysis method

To evaluate our approach, we collected a set of community smells as discussed in Section 4.3.1. Table 3 summarizes the collected smells. Furthermore, as a sanity check, all smells were manually inspected and validated based on guidelines from the literature as well as through our survey with the original developers. Furthermore, our dataset is available online for future extension and replication dat (2020).

We considered eight common types of community smells, *i.e.*, *organisational silo effect* (OSE), *black-cloud effect* (BCE), *prima-donnas effect* (PDE), *sharing villainy* (SV), *organisational skirmish* (OS), *solution defiance* (SD), *radio silence* (RS), and *truck factor* (TF), (cf. Section 3.2). In our experiments, we conducted a 10-fold cross-validation procedure to split our data into training data and evaluation data.

Table 3.3 Dataset statistics.

Data	Statistic
Number of projects	103
Number of projects having at least one smell	103
Total number of smells	407
Average number of smells per project	4.6
Average number of developers per project	229
Number of projects with <50 developers	40
Number of projects with 50 – 150 developers	34
Number of projects with >150 developers	29
Average number of commits per project	1,103
Average number of days in each project	3,233

To answer **RQ1**, we carry out a set of experiments to justify our GP-ECC approach. We first compare the performance of our meta-algorithm ECC to two well-known meta-algorithms with proven success in MLL, *random k-labelset* (RAKEL) Tsoumakas & Katakis (2007) and *binary relevance* (BR) Tsoumakas & Katakis (2007). We next used GP, *decision tree* (J48) and *random forest* (RF) as their corresponding underlying classification algorithms. We also compared with the widely used MLL algorithm adaptation method, *K-Nearest Neighbors* (ML.KNN) Zhang & Zhou (2007). Thus, in total, we have 10 MLL algorithms to be compared. One fold is used for the test and 9 folds for the training.

To compare the performance of each method, we use common performance metrics, *i.e.*, precision, recall, and F-measure Read *et al.* (2011); Ouni *et al.* (2017); Kessentini & Ouni (2017); Xia *et al.* (2014). Let l a label in the label set L. For each instance i in the smells learning dataset, there are four outcomes, True Positive (TP_l) when i is detected as label l and it correctly belongs to l; False Positive (FP_l) when i is detected as label l and it actually does not

belong to l; False Negative (FN_l) when i is not detected as label l when it actually belongs to l; or True Negative (TN_l) when i is not detected as label l and it actually does not belong to l. Based on these possible outcomes, precision (P_l), recall (R_l) and F-measure (F_l) for label l are defined as follows:

$$P_l = \frac{TP_l}{TP_l + FP_l} \quad ; \quad R_l = \frac{TP_l}{TP_l + FN_l} \quad ; \quad F_l = \frac{2 \times P_l \times R_l}{P_l + R_l}$$

Then, the average precision, recall, and F-measure of the $\mid L \mid$ labels are calculated as follows:

$$Precision = \frac{1}{\mid L \mid} \sum_{l \in L} P_l \quad ; \quad Recall = \frac{1}{\mid L \mid} \sum_{l \in L} R_l \quad ; \quad F1 = \frac{1}{\mid L \mid} \sum_{l \in L} F_l$$

Statistical test methods. To compare the performance of each method, we perform Wilcoxon pairwise comparisons Cohen (2013) at 99% significance level (*i.e.*, $\alpha = 0.01$) to compare GP-ECC with each of the 9 other methods. We also used the non-parametric effect Cliff's delta (d) Cliff (1993) to compute the effect size. The effect size d is interpreted as Negligible if $\mid d \mid < 0.147$, Small if $0.147 \leq \mid d \mid < 0.33$, Medium if $0.33 \leq \mid d \mid < 0.474$, or High if $\mid d \mid \geq 0.474$.

To answer **RQ2**, we investigated the community smell types that were detected to find out whether there is a bias towards the detection of specific smell types.

To answer **RQ3**, we aim at identifying the features that are the most important indicators of whether a project has a given community smell or not. For each smell type, we count the percentage of rules in which the feature appears across all obtained optimal rules by GP. The more a feature appears in the set of optimal trees, the more the feature is relevant to characterize that smell.

Algorithms parameters. For all the GP, RF and J48 algorithms, the maximum depth of the tree is set to 10. For GP, the population size is 200, number of iterations is 3,000, crossover and mutation rates are 0.9 and 0.1, respectively. For RF and J48, we used the default parameters of Weka. The number of neighbors of ML.KNN is set to 10. For ECC, we set the ensembles size $n = 20$. For RAKEL, we set $n = 20$, and the labels subset $k = 4$.

3.4.3 Results

Results for RQ1 (Performance). Table 4.4 reports the average precision, recall and F-measure scores for the 10 methods. We observe that ECC competes well against the other 2 meta algorithms RAKEL and BR methods. Looking at the base learning methods (GP, J48 and RF), we used GP-ECC as the base for determining statistical significance. In particular, the GP-ECC method achieves the highest F-measure with 0.89 compared to the other methods with medium and large effect sizes, except with GP-RAKEL for which the results were statistically different but with small different effect size. The same performance was achieved in terms of precision and recall, with 0.87 and 0.91, respectively. Moreover, we observe that GP-ECC achieves comparable performance as GP-RAKEL in terms of recall which confirms the suitability of the GP formulation compared to decision tree and random forest algorithms. We can also see overall superiority for GP-ECC compared to the transformation method ML.KNN in terms of precision, recall and F-measure with large effect size. One of the reasons that BR does not perform well is that it ignores the label correlation, while RAKEL and ECC consider the label correlation by using an ensemble of classifiers. Moreover, among the 3 base learning algorithms, GP performs the best, followed by decision tree (J48) and random forest (RF).

Results for RQ2 (Sensitivity). Figure 4.7 reports the sensitivity analysis of each specific community smell type. We observe that GP-ECC does not have a bias towards the detection of any specific smell type. As shown in the figure, GP-ECC achieved good performance and low variability in terms of both precision (ranging from 0.84 to 0,9) and recall (ranging from 86 to 92) across the 8 considered smell types. The highest precision and recall was obtained for the organisational silo effect (OSE), the track factor (TF), and solution defiance (SD) which heavily

Table 3.4 The achieved results by each of the meta-algorithms ECC, RAKEL and BR with their base learning algorithms GP, J48, and RF; and ML.KNN.

Algorithm	**Precision**		**Recall**		**F1**	
	score	**p-value (d)***	**score**	**p-value (d)***	**score**	**p-value (d)***
GP-ECC	**0.87**	-	**0.91**	-	**0.89**	-
J48-ECC	0.84	<0.01 (M)	0.89	<0.01 (S)	0.86	<0.01 (M)
RF-ECC	0.84	<0.01 (M)	0.87	<0.01 (M)	0.85	<0.01 (M)
GP-RAKEL	0.85	<0.01 (S)	**0.91**	No. Stat. Sig.	0.88	<0.01 (S)
J48-RAKEL	0.83	<0.01 (L)	0.88	<0.01 (M)	0.85	<0.01 (L)
RF-RAKEL	0.84	<0.01 (M)	0.86	<0.01 (M)	0.85	<0.01 (M)
GP-BR	0.83	<0.01 (L)	0.85	<0.01 (M)	0.84	<0.01 (L)
J48-BR	0.81	<0.01 (L)	0.82	<0.01 (M)	0.81	<0.01 (L)
RF-BR	0.82	<0.01 (L)	0.82	<0.01 (L)	0.82	<0.01 (L)
ML.KNN	0.82	<0.01 (L)	0.84	<0.01 (L)	0.83	<0.01 (L)

* p-value(*d*) reports the statistical difference (p-value) and effect-size (*d*) between GP-ECC and the algorithm in the current row. The effect-size (*d*) is N : Negligible – S : Small – M : Medium – L : Large

relies on the notion of developers social network and sub-groups. This higher performance is reasonable since the existing guidelines Tamburri *et al.* (2019); Ferreira *et al.* (2016); Tamburri *et al.* (2016, 2015a, 2018); Tamburri *et al.* (2019, 2013) rely heavily on the notion of social network. But for smells such as organisational skirmish (OS) and radio silence (RS), the notion of social network is less important and this makes this type of smells hard to detect using such information.

Results for RQ3 (Features influence). To better understand what features are most selected by our GP to generate detection rules among all the generated rules, we count the percentage of rules in which the feature appears. Table 3.5 shows the statistics for each smell type with the top-10 features (cf. Table 2.2), from which the three most influencing features values are in bold. We observe that the graph betweeness, closeness and degree centrality (GBC, GCC, and GDC), the network density (ND), the standard deviation of developers per community and per time zone (SDC and SDZ) as well as the number of communities (NC). We thus observe that different social network patterns play a crucial role in the emergence of community smells. These findings suggest that more attention has to be paid to these particular socio-organizational

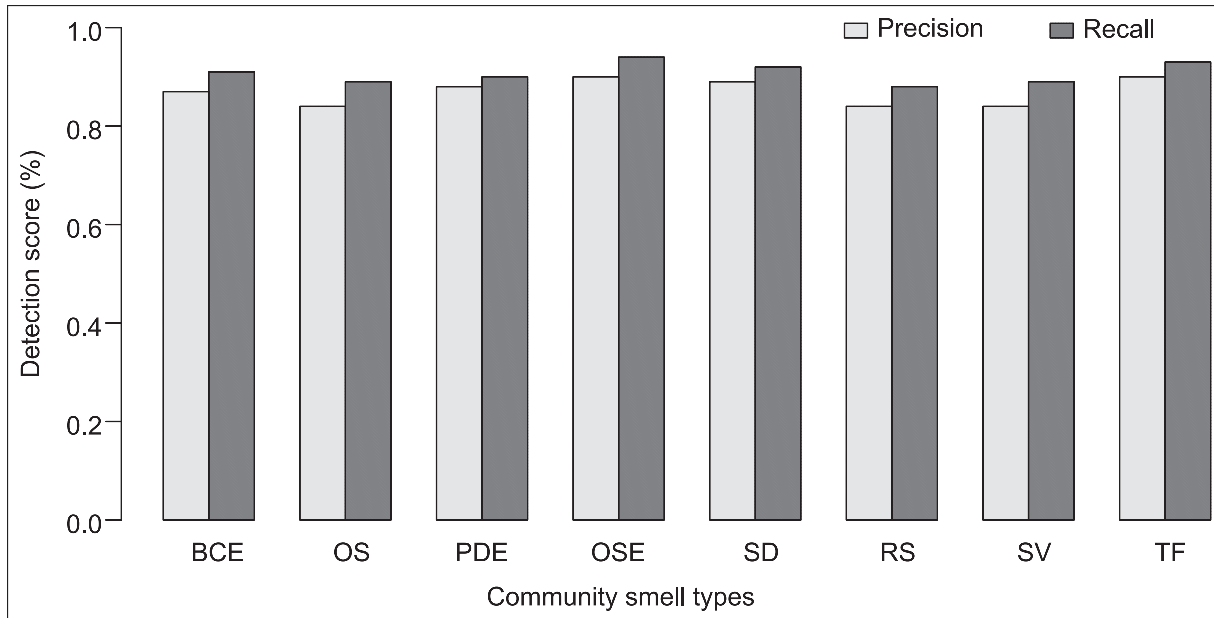


Figure 3.4 The achieved precision and recall scores by GP-ECC for each communiy smell type.

characteristics within the software project community to avoid the presence of smells and their impact on the software project.

Table 3.5 The most influential features for each smell.

metric	OSE	BCE	PDE	SV	OS	SD	RS	TF
GDC	**95**	**91**	**92**	**92**	**90**	83	**96**	**95**
GCC	91	**88**	**89**	**95**	**93**	**91**	91	**93**
SDZ	87	53	**89**	**88**	62	88	72	63
ND	**96**	81	82	**88**	**90**	**92**	92	93
GBC	**93**	**92**	**90**	71	**91**	**96**	92	92
NC	82	62	72	43	52	62	**95**	72
SDC	91	**88**	**89**	**88**	**91**	66	82	**95**
TZ	76	48	72	62	53	**92**	**97**	47
TFN	18	15	21	41	22	42	39	**100**
PSD	53	21	18	45	32	62	65	81

3.5 Threats to validity

Threats to construct validity could be related to the performance measures. We basically used standard performance metrics such as precision, recall and F-measure that are widely accepted

in MLL and software engineering Read *et al.* (2011); Ouni *et al.* (2017); Kessentini & Ouni (2017); Xia *et al.* (2014); Ouni *et al.* (2013b). Another potential threat could be related to the selection of classification techniques. Although we use the GP, J48 and RF techniques which are known to have high performance, there are other techniques. To mitigate this threat, we plan to compare with other ML techniques.

Threats to internal validity relate to errors in our experiments. We have double checked our experiments and the datasets collected following the literature guidelines and a survey with developers Tamburri *et al.* (2019); Tamburri *et al.* (2016, 2015a, 2013b), still there could be errors that we did not notice.

Threats to external validity relate to the generalizability of our results. We have analyzed a total of 407 smell instances from 103 different open source projects, different community sizes and programming languages. In the future, we plan to reduce this threat further by analyzing more projects from more industrial and open-source software projects.

3.6 Chapter Summary

We introduced in this paper an automated approach to detect community smells in software projects. We formulate the problem as a multi-label learning problem using the ECC meta-algorithm with an underlying GP model. Our GP-ECC aims at generating detection rules for each smell type. We use GP to translate regularities and symptoms that can be found in real-world community smell examples into detection rules. A detection rule is a combination of socio-technical attributes/symptoms with their appropriate threshold values to detect various types of community smells. We evaluated our approach on a set of 103 projects and 407 smell instances across 8 common types of community smells. Results show that our GP-ECC approach can identify all the considered community smell types with an average F-measure of 89% and outperforms 9 state-of-the-art MLL techniques that rely on different meta-algorithms (ECC, BR and RAKEL) and different underlying learning algorithms (GP, J48, and RF); and a transformation method ML.KNN. Moreover, we conducted a deep analysis to investigate the

symptoms, *i.e.*, features, that are the best indicators of community smells. We find that the standard deviation of the number of developers per time zone and per community, and the social network betweenness, closeness and density centrality are the most influential characteristics.

As future work, we plan to extend our approach with more open source and industrial projects to provide ampler empirical evaluation. We plan also to extend our approach to provide software project managers with community change recommendations to avoid social debt in their projects. We also plan to assess the impact of community smells on different aspects of software projects.

CHAPTER 4

IMPROVING THE DETECTION OF COMMUNITY SMELLS THROUGH SOCIO-TECHNICAL AND SENTIMENT ANALYSIS

Nuri Almarimi[a] , Ali Ouni[a] , Moataz Chouchen[a] , Mohamed Wiem Mkaouer[b]

[a] Department of Software Engineering and IT, École de Technologie Supérieure
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

[b] Rochester Institute of Technology, 1 Lomb Memorial Dr, Rochester,
NY 14623, United States

Paper published in *Journal of Software: Evolution and Process*, September 2022.

Abstract Open source software development is regarded as a collaborative activity in which developers interact to build a software product. Such a human collaboration is described as an organized effort of the "social" activity of organizations, individuals, and stakeholders, which can affect the development community and the open source project health. Negative effects of the development community manifest typically in the form of community smells which represent symptoms of organizational and social issues within the open source software development community that often lead to additional project costs and reduced software quality. Recognizing the advantages of the early detection of potential community smells in a software project, we introduce a novel approach that learns from various community organizational, social, and emotional aspects to provide automated support for detecting community smells. In particular, our approach learns from a set of interleaving organizational-social and emotional symptoms that characterize the existence of community smell instances in a software project. We build a multi-label learning model to detect 10 common types of community smells. We use the ensemble classifier chain (ECC) model that transforms multi-label problems into several single-label problems which are solved using genetic programming (GP) to find the optimal detection rules for each smell type. To evaluate the performance of our approach, we conducted an empirical study on a benchmark of 143 open-source projects. The statistical tests of our results show that our approach can detect community smells with an average F-measure of 93%

achieving a better performance compared to different state-of-the-art techniques. Furthermore, we investigate the most influential community-related metrics to identify each community smell type.

Keywords: Human factors in software engineering, Software organization and properties.

4.1 Introduction

Open Source Software (OSS) development is a collaborative activity in which developers interact between each other to develop and maintain a software system Christley & Madey (2007); Gamalielsson & Lundell (2014). Nowadays, OSS is becoming an essential part in the software economy. The number of OSS projects and their size has been growing at exponential rate (Deshpande & Riehle (2008)). The success of OSS development leads to building sustainable software systems Feitelson (2012); Gamalielsson & Lundell (2011, 2014). Unlike proprietary software, OSS projects can be developed and maintained by the whole community providing more transparency on the source code, development processes and technologies as well as developers communications. As there exist a substantial number of OSS projects, there is a continuous concern about these projects' health Palomba *et al.* (2018b); Tamburri *et al.* (2019). Understanding the characteristics of factors of healthy and thriving OSS communities is crucial to evaluate existing efforts and to identify improvement opportunities Gonzalez-Barahona, Sherwood, Robles & Izquierdo (2017); Goggins *et al.* (2021).

Recent works found that the interactions between developers not only affect OSS social debt De Stefano, Pecorelli, Tamburri, Palomba & De Lucia (2020) but also software code quality Bettenburg & Hassan (2010). Several organizational-social structures challenges can arise in OSS sub-optimal communities, (e.g., poor communication sub-teams, different cultures levels, as well as expertise Sharp, Robinson & Woodman (2000a); Jaakkola (2012); Hofstede, Jonker & Verwaart (2008); Greenhoe (2016a)). As a result, such organizational and social patterns may impact the OSS community's health and lead to *community smells* Tamburri *et al.* (2013b, 2015a); Almarimi *et al.* (2021). Community smells are introduced as sub-optimal organizational

patterns and social characteristics that may cause the emergence of social debt Tamburri *et al.* (2016, 2015a). Social debt is defined as unforeseen costs of sub-optimal organizational and social characteristics that hamper to shorten and/or lengthen the straightforward operation, production, and evolution of software. Such issues could manifest in different forms, e.g., lack of cooperation, communication or coordination within software community teams.

Moreover, collaboration artifacts of OSS projects are a place where development teams can express their emotions, in such a way that may affect their collaboration either positively or negatively Murgia *et al.* (2014). A recent study has found that emotions impact software development communities i.e., task quality, productivity, creativity, and job satisfaction De Choudhury & Counts (2013). In different cases, poor circumstances and bad practice are enforced within established software projects life cycle. Moreover, the big challenges of community smells can be found in the same project with similar symptoms. Therefore, such symptoms make the detection of community smells even harder and error-prone Avelino *et al.* (2016a); Tamburri *et al.* (2018); Damian A. Tamburri (2016); Palomba *et al.* (2018b); Tamburri *et al.* (2019). For instance, some of the common smells are *Organizational Silo Effect*(OSE) Tamburri *et al.* (2015a); Tamburri *et al.* (2019) and the *Solution Defiance*(DF) Tamburri *et al.* (2015a); Tamburri *et al.* (2019) are typically associated in the form of lack of communication and lack of cooperation due to the independent subgroups through the development team.

Detecting community smells to explore open source community's health has become important to help developers and software projects managers in picking up smells in an easier and more accurate fashion. Regardless the impact of such smells on the OSS projects and community health, they are organizational patterns that can help us understand the health of a community. Despite the effort of current works to characterize, identify, and define community smells' patterns, there is a lack of knowledge on how to define and calibrate a set of detection rules to match the symptoms of a community smell with the actual characteristics of a software project Avelino *et al.* (2016a); Tamburri *et al.* (2019c); Palomba *et al.* (2018b); Tamburri *et al.* (2021). To address these issues, we introduced an automated approach based on Genetic Programming to learn from the symptoms of real-world instances of community smells Almarimi, Ouni,

Chouchen, Saidani & Mkaouer (2020a). However, the proposed approach is mainly based on commit changes to identify developers working on the same files, and we consider two important factors (1) the communications between developers through "issues channels discussion" and "pull requests", and (2) the sentiments and emotions of developers in their communications.

In this paper, we introduce an approach to detect community smells in open source projects namely, extended Genetic Programming-based Ensemble Classifier Chain (eGP-ECC). We design the problem as a multi-label learning (MLL) problem to solve the interleaving symptoms of community smells' issues, by generating multiple detection rules that can identify various of community smells. Our technique is used to detect each smell type individually by convert the detection problem into multiple binary classification problems. Our Ensemble Classifier Chain (ECC) technique Read *et al.* (2011) involves the training of n single-label binary classifiers, where each one is solely responsible for detecting a specific label, i.e., community smell type. These n classifiers are linked in a chain, such that each binary classifier is able to consider the labels identified by the previous ones as additional information at classification time. For the binary classification, we exploit the effectiveness of genetic programming (GP) John R. Koza (1992); Glover & Kochenberger (2006); Deb *et al.* (2002); Ouni *et al.* (2017); Kessentini & Ouni (2017); Ouni *et al.* (2013b) to find the optimal detection rules for each community smell. The goal of GP is to learn detection rules from a set of real-world instances of community smells. In fact, we use GP to translate regularities and symptoms that can be found in real-world community smell examples into detection rules. A detection rule is a combination of socio-technical attributes/symptoms and their relative threshold values, in order to detect various types of community smells.

Moreover, to better capture and analyze developers communications, we build a developer's social network based on two important communication channels including (1) issues and (2) pull requests discussion channels. Furthermore, we incorporate sentiments and emotions analysis to analyze developers communications.

We implemented and evaluated our approach on a benchmark of 143 open source projects hosted in GitHub. We first conducted a survey with developers to validate the identified instances of community smells found in the studied projects. Evaluating the performance of our extended GP-based Ensemble Classifier Chain approach (referred as the eGP-ECC hereinafter), the statistical analysis of our results shows that the generated detection rules can identify the ten considered community smell types with an average F-measure of 93% and outperforms state-of-the-art MLL techniques. Moreover, we conducted a deep analysis to investigate the symptoms, *i.e.*, features, that are the best indicators of community smells. We find that the standard deviation of the number of developers per time zone and per community, and the social network betweenness, closeness and density centrality and the ratio of issues with negative sentiments,anger words and polite comments in PR and issue discussions are the most influential characteristics.

4.1.1 Novelty statement

In this paper, we build in top of our previous work published in the 15th IEEE/ACM International Conference on Global Software Engineering Almarimi *et al.* (2020a) for community smells detection and extend it in the following ways:

1. We extend our study with two additional and common types of community smells, namely, unhealthy interaction (UI) Tourani *et al.* (2014); Mäntylä, Adams, Destefanis, Graziotin & Ortu (2016), and Unfriendly communication (UC) Raman *et al.* (2020); Murgia *et al.* (2018); Ortu *et al.* (2015b).
2. We extend our developers social network with common communication channels from (*i*) pull requests, and (*ii*) issues reports to better capture direct communications between developers in a project.
3. We extend our metrics suite to better capture the different symptoms of community smells by considering additional (*i*) category of socio-technical characteristics based on sentiments analysis such as sentiments polarity, politeness and anger emotions to better capture

community smell symptoms, and (*ii*) developers communication and productivity metrics based on pull requests and issues tracking related information.

4. We extend our experimental setup and dataset with (*i*) the newly studied community smells, (*ii*) new developers social network based on pull requests and issue reports, and (*iii*) the new features based on sentiments analysis dat (2020).
5. We extend the related work to cover more aspects of community smells, OSS projects health and sentiments analysis.

4.1.2 Replication package

Our comprehensive replication package is available online for future replications/extensions dat (2020).

4.1.3 Paper organization

The remainder of the rest paper is organized as follows. Section 4.2 presents the necessary background. In section 4.3, we discuss our eGP-ECC approach for community smells detection. Section 4.4 provides our empirical evaluation, and discusses the obtained results. Section 4.5 examines the threats to validity. Finally, in Section 4.6, we conclude and outline our future work.

4.2 Background

4.2.1 Community Smells in Open Source Software

Community smells are defined as a set of social-organizational circumstances that occur within the software development community, having a negative effect on the relations health within the development community which may cause social debt over time Tamburri *et al.* (2015a); Tamburri & Palomba (2021); Tamburri *et al.* (2018); Avelino *et al.* (2016a). A number of community smells have been defined in the literature. We refer to the following community

smell types as theoretical lens to explore open source community health Tamburri *et al.* (2015a); Avelino *et al.* (2016a):

- **Organizational Silo Effect (OSE):** The OSE smell is manifested in OSS project when too high decoupling between developers, isolated subgroups, and lack of communication and collaboration between community developers occur. The consequence of this smell is an extra unforeseen cost to a project by wasted resources (*e.g.*, time), as well as duplication of code Tamburri *et al.* (2015a, 2016, 2018).
- **Black-cloud Effect (BCE):** The BCE smell occurs when developers have a lack of information due to limited knowledge sharing opportunities (*e.g.*, collaborations, discussions, daily stand-ups, etc.), as well as a lack of expert members in the project that are able to cover the experience or knowledge gap of a community. The BCE may cause mistrust between OSS community members and creates selfish behavioral attitudes Tamburri *et al.* (2015a); Tamburri *et al.* (2019); Tamburri *et al.* (2018).
- **Prima-donnas Effect (PDE):** The PDE smell occurs when a team of people is unwilling to respect external changes from other team members due to inefficiently structured collaboration within the community. The presence of this smell may affect OSS community health and create isolation problems, superiority, constant disagreement, uncooperativeness, and raise selfish team behavior, also called "prima-donnas" Tamburri *et al.* (2015a, 2016); Goggins *et al.* (2021).
- **Sharing Villainy (SV):** The SV smell is caused in OSS community by a lack of high-quality information exchange activities (*e.g.*, face-to-face meetings). The main side effect of this smell limitation is that community members share essential knowledge such as outdated, wrong and unconfirmed information Tamburri *et al.* (2015a); Sen, Singh & Borle (2012).
- **Organizational Skirmish (OS):** The OS smell is caused by a misalignment between different expertise levels and communication channels among development units or individuals involved in OSS project. The existence of this smell often leads to dropped productivity and affects the project's timeline and cost Tamburri *et al.* (2015a); Jansen (2014); Tantisuwankul *et al.* (2019).

- **Solution Defiance (SD):** The solution defiance smell occurs when the OSS development community presents different levels of cultural and experience background, and these variances lead to the division of the community into similar subgroups with completely conflicting opinions concerning technical or socio-technical decisions to be taken. The existence of the SD smell often leads to unexpected project delays and uncooperative behaviors among the developers Tamburri *et al.* (2015a); Rastogi (2016).
- **Radio Silence (RS):** The RS smell occurs when a high formality of regular procedures takes place due to the inefficient structural organization of a community. The RS community smell typically causes changes to be retarded, as well as a valuable time to be lost due to complex and rigid formal procedures. The main effect of this smell is an unexpected massive delay in the decision-making process due to the required formal actions needed Tamburri *et al.* (2015a); Rastogi (2016); Tsay, Dabbish & Herbsleb (2014).
- **Truck Factor (TF):** It occurs in OSS community when most of the project information and knowledge is concentrated in one or few developers. The presence of this smell eventually leads to a significant knowledge loss due to the turnover of developers Ferreira *et al.* (2016); Avelino *et al.* (2016a); Sen *et al.* (2012).
- **Unhealthy Interaction (UI):** This smell occurs when discussions between developers are slow, light, brief and/or contains poor conversations. It manifests with low developers participation in the project discussions (*e.g.*, pull requests, issues, etc.) having long delays between messages communications Tourani *et al.* (2014); Mäntylä *et al.* (2016). Such unhealthy interactions may impact the project (*e.g.*, increased issues resolution time, pull request acceptance), and demotivate and burn out developers, which can in turn create challenges for sustaining the project. This smell also leads to individual contributions or selfish characteristics of developers within OSS project.
- **Unfriendly Communication (UC):** This smell occurs when communications between developers are subject to negative sentiments containing unpleasant, anger or even conflicting opinions towards various issues that people discuss. Developers in OSS project may have negative interpersonal interactions with their peers, which express feelings of sadness towards

a problem. These negative interactions may ultimately result in developers abandoning projects Raman *et al.* (2020); Murgia *et al.* (2018); Ortu *et al.* (2015b); Tourani *et al.* (2014).

In this paper, we focus primarily on these smells as they are widely studied and most occurring in the software industry as well as in open-source projects based on recent studies Tamburri *et al.* (2016, 2015a); Palomba *et al.* (2018b); Ferreira *et al.* (2016); Almarimi *et al.* (2020a); Almarimi, Ouni & Mkaouer (2020b); Almarimi *et al.* (2021).

4.2.2 Search Based Software Engineering

Recently Search-Based Software Engineering (SBSE) has attracted attention to solving software engineering challenges that required a search for near-optimal or optimal solutions Harman & Jones (2001). SBSE has presented new ways, which transform software engineering problems from human-based search to machine-based search techniques Harman & Jones (2001); Harman (2007). Thus, the use of SBSE mainly on heuristics that follow the automated search to avoid the boring human-in-the-loop search tasks. The main goal of this technique is to formulate software engineering problems into search problems through define activities of suitable solution representation, fitness function, and solution change operators. However, the powerfully of this approach has many of search algorithms involve single to many-objective techniques that can be used to solve the problem Harman *et al.* (2012b); Ouni *et al.* (2016a,b, 2013a,c, 2017, 2012). In this work, we introduce the community smells detection problem as a SBSE problem. Therefore, we applied genetic programming (GP) to solve our problem and generate detection rules with ECC.

4.2.3 Multi-Label Learning

Multi-Label Learning (MLL) is the classification technique that use machine learning approach to move the problem into multi-class classification problem de Carvalho & Freitas (2009); Tsoumakas *et al.* (2010a); Tsoumakas & Katakis (2007); Read *et al.* (2011). MLL has been applied in several areas that assigned to multi-label classification problem using machine

learning and data mining Tsoumakas & Katakis (2007). The existing of the MLL techniques and algorithms including (i) Classifier Chain (CC) Read *et al.* (2011), Label Powerset (LP), and the Binary Relevance (BR) algorithm that used as transformation methods Tsoumakas & Katakis (2007) (ii) the K-Nearest Neighbors (ML.KNN) algorithm that use as adaptation methods Zhang & Zhou (2007) (iii) The Ensemble Classifier Chain (ECC) algorithm and RAndom K-labELset (RAKEL). MLL algorithms has been applied successfully as ensemble methods to solve different search problems McIlroy *et al.* (2016); Podgurski *et al.* (2003); Feng & Chen (2012); Xia *et al.* (2014).

The Classifier Chain (CC) model. The CC model combines the computational efficiency of the BR method while still being able to take the label dependencies into account for classification. With BR, the classifier chains method involves the training of q single-label binary classifiers and each one will be solely responsible for classifying a specific label $l_1, l_2, ..., l_q$. The difference is that, in CC, these q classifiers are linked in a chain $\{h_1 \rightarrow h_2 \rightarrow \ ... \rightarrow h_q\}$ through the feature space. That is, during the learning time, each binary classifier h_j incorporates the labels predicted by the previous $h_1, ..., h_{j-1}$ classifiers as additional information. This is accomplished using a simple trick: in the training phase, the feature vector x for each classifier h_j is extended with the binary values of the labels $l_1, ..., l_{j-1}$.

The Ensemble Classifier Chain (ECC) model was introduced to improve the CC model limitations and solve the problem of random labels and may lead to being poorly ordered. Furthermore, this issue course an inaccurate detection when one or more beginner's labels predict poorly Read *et al.* (2011). To mitigate the issue mentioned, the ensemble of chains was applied with a random label order to avoid a negative impact on classification accuracy. The best solution is selected by a majority voting method. In addition, the approach of ensembles has been applied successfully in search problems and achieve high predictive performance Read *et al.* (2011).

Furthermore, there exists various other traditional classifiers for MLL in the literature including:

RAndomk-labELsets (RAKEL) transforms multi-label problems into several single-label problems, and considers label correlations using single-label classifiers that are trained using a different small random subset of the set of labels Tsoumakas & Katakis (2007).

Binary Relevance (BR) transforms a multi-label problem into multiple binary problems; for each problem one label, and adopted without considering correlations among the labels Tsoumakas & Katakis (2007).

Multi-label K-nearest neighbor (ML-KNN) is a multi-label lazy learning algorithm that identify the label set of the test instance using the maximum a posteriori principle, based on prior and posterior probabilities for the frequency of each label within the k nearest neighbours Zhang & Zhou (2007). In this work, we build the ECC method that linked the lack between MLL and SBSE and applied it to the community smells detection problem, where a single project may contain several types of community smells with mutual symptoms.

For the binary classifications, the ECC model adopts a search-based approach using genetic programming (GP) to learn detection rules for each smell type. To gain a deeper understanding, we present in Figure 4.1 a real word example of SD smell. The SD smell occurs with different levels of cultural and background diversity within the software development community, and these variances lead to the division of the community into similar subgroups. This SD smell could be detected by features such as graph closeness centrality (GCC), graph degree centrality (GDC), standard deviation of a graph degree centrality in a project (SDD), average number of comments per issue report (ANCI), ratio of issues with negative sentiments (RINC), Formal network (FN), The average number of authors per issue report (ANAI), and standard deviation of authors per issue report (SDAI). Figure 4.1 presents social-technical metrics along with threshold values. These values can be used as conditions to provide software project managers with community change recommendations, helping them avoid social debt in their projects.

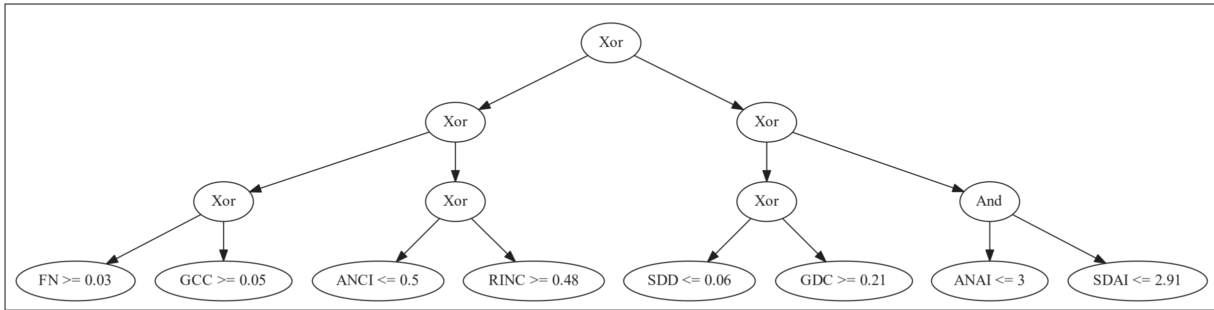


Figure 4.1 A real-world example of SD smell.

4.3 Approach

Our approach consists of identifying a set of detection rules to match the symptoms of a community smell with the actual characteristics of a given software project, rather than relying on manual effort and human expertise Tamburri *et al.* (2013). The main *goal* of our approach is to infer a set of detection rules for each community smell type while taking into consideration the dependencies between the different smell types and their interleaving symptoms.

Figure 4.2 presents an overview of our approach to generating community smells detection rules using the GP-ECC model. Our approach consists of two phases: training phase and detection phase. In the training phase, our goal is to build an ensemble classifier chain (ECC) model learned from real-world community smells identified from software projects based on several GP models for each individual smell. In the detection phase, we apply this model to detect the proper set of labels (*i.e.*, types of community smells) for new unlabeled data (*i.e.*, a new project).

Our framework takes as inputs a set of software projects with known labels, *i.e.*, community smells (phase A). Then, extracts a set of features characterizing the considered community smell types from which a GP algorithm will learn (phase B). Next, an ECC algorithm will be built (phase C). The ECC algorithm consists of a set of classifier chain models (CC), each with a random label order. Each CC model, learns ten individual GP models for each of the ten considered smell types. The ith binary GP detector will learn from the training data while considering the existing i already detected smells by the $i-1$ detected smells generate the

optimal detection rule that can detect the current i^{th} smell. In total, the ECC trains n multi-label CC classifiers $CC_1, ..., CC_n$; each classifier is given a random chain ordering; each CC builds 10 binary GP models for each smell type. Each binary model uses the previously predicted binary labels into its feature space. Then, our framework searches for the near optimal GP-ECC model from these n multi-label chain classifiers using an ensemble majority voting schema based on each label confidence Read *et al.* (2011).

In the detection phase, the GP-ECC model obtained from the training phase that assigns multiple labels, *i.e.*, community smells types, to a new project based on its current features, *i.e.*, its socio-technical characteristics (step C). In the next subsections, we provide the details of each step.

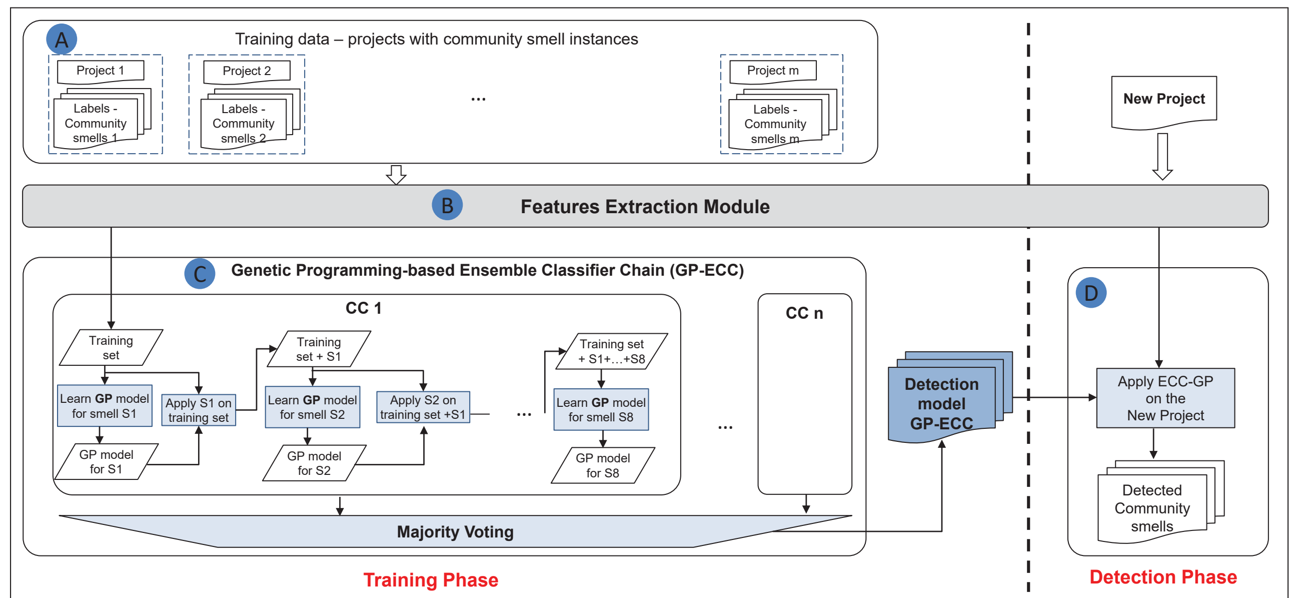


Figure 4.2 The community smells detection framework using the GP-ECC model.

4.3.1 Step A : Training data collection

An important step to solving the problem of community smells detection is to prepare the learning dataset. To build a base of real-world community smell examples that occur in software projects, we selected a set of software projects which are diverse in nature (*e.g.*, size, application

domains, etc.) that have experienced community smells. We considered a set of open-source projects from GitHub to access their development history. The considered filters to collect the training data are the following:

- *Commit size*: the projects vary from medium size >10 KLOC, large size (10 – 60) to very large size < 60 KLOC.
- *Community size*: the projects team size vary from medium (<100 members), large (100-900), to very large (> 900).
- *Programming language*: the selected projects are implemented in different programming languages, including Java, C#, Python and C.
- *Availability of communication channels*: the selected projects use (1) pull requests and (2) issue tracking systems hosted in their GitHub repositories.
- *Existence of community smells:* the project should contain at least one instance of community smell.

Table 4.1 Questionnaire filled in by the study participants.

Part 1: Background Information:	
1	What is your project name on GitHub ? * [Required Answer]
2	How do you describe your occupation in this project? [Student −Part-time employee − Full-time employee −Unemployed −Retired − Other]
3	How long time you have been working on open source projects? [Less than 3 years −3 years or more −Other]
Part 2: Social aspects perception	
4	Do you think there is a lack of communication and cooperation within the development team (share knowledge, experiences, exchange activities, etc.) in your project?
5	Do you think there is a delay in communications between developers in your project?
6	Do you think there is/are developer(s) in your project working on the same code and communicating by means of a third person?
7	Do you think there is/are in your project developer(s) who have selfish and/or egoistical behaviors?
8	Do you think there are different communication and expertise levels that may cause misalignment within the development team in your project?
9	Do you think there are different levels of cultural and experiential backgrounds in the development team?
10	Do you think there is/are isolated subgroup(s) within the development team of your project?
11	Do you think there is in your project a formal and complex organizational structure (with potentially unnecessary “regular procedures")?
12	Do you think there is in your project some unique knowledge and information brokers toward different developers?
13	Do you think there is in your project risk that a core developer can unexpectedly leave the project and can influence the development process?
14	Do you think there is in your project a waste of resources (e.g., time) over the development life-cycle?

To collect our base of examples, the authors checked individually all identified community smells if they match with the state-of-the-art definitions, characteristics and symptoms. All community

smell instances that did not reach a full agreement were excluded from our base of examples. After the manual inspection of potential existence of community smells, we finally ended with 143 from an initial set of 160 projects projects that have diverse types of community smells dat (2020). As an attempt to validate our identified smells, we conducted a survey with the original developers of the selected projects to get their feedback by following an opt-in strategy Hunt *et al.* (2013) for our survey. While the survey will help us to validate our identified symptoms, it can also help to understand whether developers are conscious of the presence of such smells in their projects. We extracted from GitHub the email address of the developers who have acted in a project at least 30 commit changes during the last 12 months and participated in the project in the last 3 years. In such a way, we focused only on developers having adequate experience with the project's community Sugar (2014). Before we sent the survey's questions, we first sent an email asking permission to participate in our study. As a result, we obtained a positive response from 62 out of 432 developers (14%) who were later contacted with the actual survey. We received answers from developers of 31 different projects in the dataset, which covered 29.3% of all considered smells. In our study, we are aware that the different opinions on this survey may not be necessarily generalized, but this analysis helped us to confirm whether our smells symptoms analysis match with the participants perception of such smells in their projects.

The survey's link was sent via email to all participants with a brief introduction. The list of questions is divided into two main parts as shown in Table 4.1. The first part consists of three control questions on the project name and background information about the participants occupation and experience with the project. Then, we asked developers to rate the validity of 11 statements that we extracted from the community smells definitions (cf. Section 4.2) using a 5 point Likert scale Likert (1932) ranging between "*Strongly Disagree*" to "*Strongly Agree*". This part presents typical situations in which community smells occur without mentioning the term social debt or smells in the questionnaire. To avoid any possible bias in the responses, we did not ask the developers directly to rate how healthy was their community. For instance, the statement "*Do you think there is a lack of communications and cooperation within the development team (share knowledge, experiences, exchange activities, etc.) in your project?*"

was aimed at understanding whether developers actually recognize the presence of the symptoms of an OSD smell in their project. The complete questions list of our survey is presented in Table 4.1. After collecting the participants answers, we compared their answers with the smell instances detected manually in our base of examples to make sure that they match. Overall, 8 smells did not match the manual analysis, and therefore we excluded them to avoid potential noise in our experimental dataset.

4.3.2 Step B: Features Extraction Module

To capture community smells symptoms, we rely on a set of metrics defined in previous studies Avelino *et al.* (2016a); Damian A. Tamburri (2016); Tamburri *et al.* (2019); Nagappan *et al.* (2008a); Pinzger *et al.* (2008a); Nordio *et al.* (2011), as well as a new set of generated metrics to capture more community-related proprieties that can be mined from the projects history. These metrics analyze different aspects in software development communities including organizational dimensional, social networks characteristics, developers collaborations, and truck numbers. Table 4.2 depicts our list of considered features. Our proposed metrics extend existing metrics to provide more details including developers social network, community structures, geographic dispersion, and developer network formality. For instance, the geographic dispersion metrics, *e.g.*, the *average number of commits per time zone* and the *standard deviation of developers per time zone* would capture the distribution of commits and developers per time zone. Our features extraction module calculates the set of metrics from a given project by analyzing its repository through commit information history available in its version control system, *e.g.*, GitHub. Figure 4.3 depicts an overview of our features extraction module which consists of two main steps to extract metrics.

Step 1. Mine developers aliases: The author alias mining and consolidation consist of the following sub-steps Avelino *et al.* (2016a): (*i*) retrieval all unique dev-emails, where for each git commit has a dev-email associated with it, (*ii*) Retrieval of GitHub logins related to developers, (*iii*) similarity matching of emails and logins: by applying Levenshtein distance Avelino *et al.* (2016a), all aliases are compared and, with a certain degree of threshold value at most one,

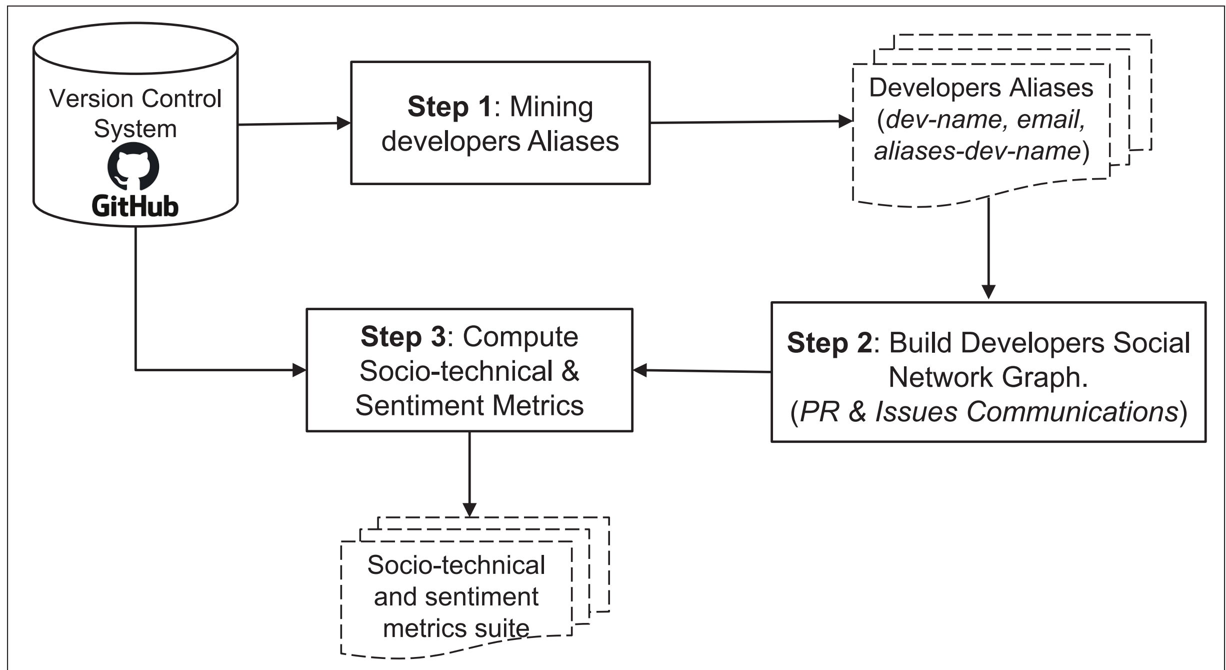


Figure 4.3 An overview of the features extraction module.

consolidated, and (*iv*) replacement of author emails by their respective aliases Avelino *et al.* (2016a). The final transformation goes through all the commits once again and replacing original authors with their main alias. As a result, if there is a developer associated with commits with different names, we consider them as a single developer, and the output will be presented in a new aliases list. For example, “Bob.Rob" and “Bob Rob" are different names for a single developer associated with commits. Thus, we consider them as the same developer in a new aliases list as a single identical substitution.

Step 2. Build a social network graph: Social network analysis (SNA) has been used to study and analyze the collaboration and communications among developers who are working in teams within software projects Meneely & Williams (2011b). Our developers social network model is based on various socio-technical communications during their software project development. Our approach builds a developer network from the version control system by tracking the change logs at three levels (1) commits, (2) issues, and (3) pull requests. Our adopted developers networks are presented as a graph in which nodes represent individual developers and the

edges are the connections between two developers, when they have participated in the same issue or in the same pull request. Moreover, our developers social network is augmented by considering connections between two developers who are working on the same files for which they make a version control commit within one month of each side Almarimi *et al.* (2020a). Such social networks allow the calculation of various communication-based metrics such the degree centrality, closeness centrality, network density, etc. (cf. Table 4.2).

Step 3. Compute socio-technical and sentiment metrics: In this step, we use the project's version control system and the extracted developers social network graph to compute a variety of socio-technical metrics as described in Table 4.2. Moreover, we considered various sentiment metrics using the state-of-the-art SentiStrength Thelwall, Buckley & Paltoglou (2012) tool [1], a widely used tool to estimate the degree of positive and negative sentiments in short texts Raman *et al.* (2020); Lin *et al.* (2018); Guzman *et al.* (2014); Murgia *et al.* (2018); Ortu *et al.* (2015b). It is a lexical sentiment extraction tool based on a list of words. We used SentiStrenght to measure the sentiments polarity of developers in commits, issues and pull request comments. SentiStrength assigns various scores for sentiment two polarizations in the interval [-5,5] as follows:

- Negative: -1 (slightly negative) to -5 (extremely negative).
- Positive: 1 (slightly positive) to 5 (extremely positive).

Moreover, we used the Stanford's Politeness detector tool Danescu-Niculescu-Mizil *et al.* (2013b); Ribeiro, Singh & Guestrin (2016) to estimate the politeness level in the developers communications. We also used the Google's Perspective API AI which scores the perceived impact that a comment might have on a conversation and identifies whether the comment could be perceived as toxic to a discussion. Finally, to better capture poor communication symptoms, we used the LIWC lexicon Tausczik & Pennebaker (2010) that detects anger conversations.

Our full list of socio-technical and sentiment metrics is described in Table 4.2. In particular, our extended metrics suite can be summarized as follows: we introduced (1) a set of 6 new

[1] http://sentistrength.wlv.ac.uk/

developer's contribution metrics based on pull requests and issue contributions (cf. developer contribution metrics in Table 4.2), (2) a set of 5 new communication metrics based on pull requests and issues discussions (cf. communication metrics in Table 4.2), and (3) a set of 9 sentiment-based metrics (cf. Sentiment Analysis metrics in Table 4.2).

Table 4.2 Socio-technical metrics framework.

Dimension	ID	Definition	Ref.
Developer Contributions metrics	NoD	*Number of developers (NoD)*: the total number of developers who have changed the code in a project.	Nagappan *et al.* (2008a)
	NAD	*Number of Active Days of an author on a project (NAD)* : the percentage of the total number of active days for each developer with respect to a project's lifetime on the total number of developers in a project.	Nagappan *et al.* (2008a)
	NCD	*Number of Commits per Developer in a project (NCD)* : the total number of times that the code has been changed by a developer with respect to the total number of commits and the total number of developers in a project.	Nagappan *et al.* (2008a)
	SDC	*Standard Deviation of Commits per developer in a project (SDC)* : the standard definition of commits per developer in a project. It provides a view of the distribution of the developers contributions.	Nagappan *et al.* (2008a)
	NCD	*Number of Core Developers (NCD)*: the total number of core developers in a project. A developer is considered as a core community member if he/she has a larger degree than peripheral developers within the developer's social network.	Tamburri *et al.* (2019)
	PCD	*Percentage of Core Developers (PCD)*: the percentage of core developers with respect to the total number of developers.	Tamburri *et al.* (2019)
	NSD	*Number of Sponsored Developers (NSD)*: the total number of sponsored developers in a project. We consider a developer that holds a sponsored status if at least 90% of her/his commits are performed during weekdays and the working day time (8am-6pm).	Tamburri *et al.* (2019)
	PSD	*Percentage of Sponsored Developers (PSD)*: the percentage of sponsored developers over the total number of developers.	Tamburri *et al.* (2019)
	NPR	*Total number of Pull Requests (NPR)*: The total number of pull requests (PR) (closed and open) in a project.	New
	SAPR	*Standard deviation of authors per PR (SAPR)*: Standard deviation of the authors count per PR in a project.	New
	ANAP	*Average number of authors per PR (ANAPR)*: The average number of authors per pull request in a project.	New
	NIS	*Number of Issues (NI)*: The total number of issues in a project.	New
	SDAI	*Standard deviation of authors per issue report (SDAI)*: Standard deviation of authors per issue in a project.	New
	ANAI	*The average number of authors per issue report (ANAI)*: The average number of authors that contribute in issue reports.	New
Social Network Analysis metrics	GDC	*Graph Degree Centrality (GDC)*: the number of connections that a developer has. The more connections with others a developer has, the more important the developer is.	Pinzger *et al.* (2008a)
	SDD	*Standard Deviation of a graph Degree centrality in a project (SDD)* : the standard deviation of the degree centrality (DC) of each developer in a project. It provides a view of the distribution of DC in a project.	Tamburri *et al.* (2019)
	GBC	*Graph Betweenness Centrality (GBC)*: a measure of the information flow from one developer to another and devised as a general measure of social network centrality. It represents the degree to which developers stand between each other. A developer with higher BC would have more control over the community as more information will pass through her/him.	Pinzger *et al.* (2008a)
	GCC	*Graph Closeness Centrality (GCC)*: a measure of the distance between a developer to other developers in the network. This metric is strongly influenced by the degree of connectivity of a network.	Pinzger *et al.* (2008a)
	ND	*Network Density (ND)*: a measure of a social network as a dense or sparse graph.	Tamburri *et al.* (2019)
Community metrics	NC	*Number of Communities (NC)*: the total number of communities in a project.	Tamburri *et al.* (2019)
	ACC	*Average of Commits per Community (ACC)*: the average number of commits per community in a project.	New
	SCC	*Standard deviation of Commits per Community (SCC)*: the standard deviation of commits performed by each community with respect to the total number of commits in a project.	New
	ADC	*Average number of Developers per Community (ADC)*: the average number of developers per community in a project with respect to the total number of developers in a project.	New
	SDC	*Standard deviation of Developers per Community (SDC)*: the standard deviation of commits performed by each community with respect to the total number of commits in a project.	New
Geographic Dispersion metrics	TZ	*Number of Time Zones (TZ)*: the total number of different time zones of developers in a project.	Nordio *et al.* (2011)
	ACZ	*Average of Commits per time Zone (ACZ)*: the average number of commits per time zone in a project.	New
	SCZ	*Standard deviation of Commits per time Zones (SCZ)*: the standard deviation of commits performed in each time zone with respect to the total number of commits in a project.	New
	ADZ	*Average number of Developers per time Zone (ADZ)*: the average number of developers per time zone in a project.	New
	SDZ	*Standard deviation of Developers per time Zones (SDZ)*: the standard deviation of developers per time zones in a project.	New
Formality metrics	NR	*Number of Releases in a project (NR)*: the total number of releases delivered in a project.	A. Gopal & S. Krishnan (2002)
	PCR	*Parentage of Commits per Release (PCR)* : the percentage of commits of each release over the total number of releases in a project.	New
	SCR	*Standard deviation of Commits per Release (SCR)* : the standard deviation of developers per release in a project.	New
	FN	*Formal Network (FN)*: the number of milestones assigned to the project with respect to the lifetime of the project.	Damian A. Tamburri (2016)
	ADPR	*Average number of days per PR (ADPR)* : the average number of days since a pull request is opened and closed in a project.	New
	ADI	*Average number of days per issue report (ADI)*: the average number of days since an issue report is opened and closed in a project.	New
Truck Number and Community Members metrics	BFN	*Bus Factor Number (BFN)*: the percentage of active core developers present in a project with respect to the total number of developers.	Cosentino *et al.* (2015)
	TFN	*Global Truck Number (GTN)*: the ratio of non-core developers in a project which can be unexpectedly lost, *i.e.*, hit by a truck before the project is discontinued.	Avelino *et al.* (2016a)
	TFC	*Truck Factor Coverage (TFC)*: the percentage of core developers and their associated authored files in a project.	Avelino *et al.* (2016a)
Communication metrics	ANCPR	*Average number of comments per PR (ANCPR)*: The average number of comments per pull request in a project.	New
	SCPR	*Standard deviation of commits per PR (SCPR)*: Standard deviation of the number of commits per PR in a project.	New
	NCI	*Number Comments in issues (NCI)*: The total number of comments in issues in a project.	New
	ANCI	*Average number of comments per issue report (ANCI)*: The average number of comments found in issues.	New
	SDCI	*Standard Deviation of Comments Count per issue report (SDCI)*: Standard deviation of Comments per issue in a project.	New
Sentiment Analysis metrics	RTCPR	*Ratio of toxic comments in PR discussions (RTCPR)*: Ratio of toxic comments in PRs with respect to the total number of PR comments as per Google's Perspective API AI.	New
	RTCI	*Ratio of toxic comments in issue discussions (RTCI)*: Ratio of toxic comments in issues with respect to the total number of issue comments as per Google's Perspective API AI.	New
	RPCPR	*Ratio of polite comments in PR discussions (RPCPR)*: Ratio of politeness in PR comments with respect to the total number of PR comments as per the Stanford's politeness detector Danescu-Niculescu-Mizil *et al.* (2013b).	New
	RPCI	*Ratio of polite comments in issue discussions (RPCI)*: Ratio of politeness in issue comments with respect to the total number of issue comments as per the Stanford's politeness detector Danescu-Niculescu-Mizil *et al.* (2013b).	New
	RINC	*Ratio of issues with negative sentiments (RINC)*: The ratio of the number of issues having negative sentiments in the comments over the total number of issues in a project. An issue is considered to be negative sentiment if the number of comments with negative sentiments is higher than 50% of the total number of comments as per the SentiStrength tool Thelwall *et al.* (2010).	New
	RNSPRC	*Ratio of negative sentiments in PR comments (RNSPRC)*: The ratio of the number of PR having negative sentiments in the comments over the total number of PRs in a project. A PR is considered to be negative sentiment if the number of comments with negative sentiments is higher than 50% of the total number of comments as per the SentiStrength tool Thelwall *et al.* (2010).	New
	RAWPR	*Ratio of anger words in PR discussions (RAWPR)*: Ratio of anger words in PR discussions with respect to the total number of words in PR comments as per the LIWC lexicon Tausczik & Pennebaker (2010).	New
	RAWI	*Ratio of anger words in PR discussions (RAWI)*: Ratio of anger words in issue discussions with respect to the total number of words in issue comments as per the LIWC lexicon Tausczik & Pennebaker (2010).	New
	ACCL	*Average Communication Comments Length (ACCL)*: The average length of comments in the developer's communication channels (PR communication issues).	New

Algorithm 4.1 High-level pseudo code of the adopted MOGP

```
1: Create an initial population P_0
2: Create an offspring population Q_0
3: t = 0
4: while stopping criteria not reached do
5:     R_t = P_t ∪ Q_t
6:     F = fast-non-dominated-sort(R_t)
7:     P_{t+1} = ∅ and i = 1
8:     while | P_{t+1} | + | F_i | ⩽ N do
9:         Apply crowding-distance-assignment(F_i)
10:        P_{t+1} = P_{t+1} ∪ F_i
11:        i = i + 1
12:    end while
13:    Sort(F_i, ≺ n)
14:    P_{t+1} = P_{t+1} ∪ F_i[N− | P_{t+1} |]
15:    Q_{t+1} = create-new-pop(P_{t+1})
16:    t = t+1
17: end while=0
```

4.3.3 Step C: Genetic Programming-based Ensemble Classifier Chain (GP-ECC)

As explained earlier in Sections 3.3.2 and 4.2.3, our approach is based on the ECC method Read *et al.* (2011) that transforms the multi-label learning task into multiple single-label learning tasks. Our multi-label ECC model aims at building a detection model to detect different instances of community smells in a software project. Each classifier chain (CC) builds a GP model for each smell type while considering the previously detected smells (if any), *i.e.*, each binary GP model uses the previously predicted binary labels into its feature space. Our choice for GP is motivated by the high performance of GP in solving challenging software engineering problems including design defects, code smells and anti-patterns detection Kessentini & Ouni (2017); Ouni *et al.* (2017); Ouni *et al.* (2013b, 2015).

In our approach, we adopted Multi-objective Genetic Programming (MOGP) as a search algorithm to generate smells detection rules. MOGP is a powerful and widely used evolutionary algorithm that extends the generic model of GP learning to the space of programs. Unlike other evolutionary search algorithms, in MOGP, solutions are themselves programs following a

tree-like representation instead of fixed-length linear string formed from a limited alphabet of symbols John R. Koza (1992).

As described in Algorithm 4.1, MOGP starts by randomly creating an initial population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_0 of size N (line 5). *Fast-non-dominated-sort* Deb *et al.* (2002) is the technique used by MOGP to classify individual solutions into different dominance levels (line 6) Deb *et al.* (2002). The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0. Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When MOGP has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance Deb *et al.* (2002) to make the selection (line 9). This parameter is used to promote diversity within the population. The front F_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of F_i are chosen (line 14). Then, a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4).

We describe in the following subsections the three main adaptation steps: solution representation, the generation of the initial generation, fitness function, and change operators.

***(i)* Solution representation.** A solution consists of a rule that can detect a specific type of community smells in the form of IF-THEN:

In MOGP, a solution is represented as a tree composed of terminals and functions. The terminals correspond to different socio-technical specific features (cf. Table 4.2) with their threshold values. The functions that can be used between these metrics are logic operators OR (union),

AND (intersection), or XOR (eXclusive OR). A solution is represented as a binary tree such that: each leaf-node (Terminal) contains one of the metrics described in Table 4.2 and their corresponding threshold values generated randomly. Each internal-node (Functions) belongs to the Connective (logic operators) set $C = \{AND, OR, XOR\}$. The threshold values are selected randomly along with the comparison and logic operators. Figure 4.4 shows a simplified example of a solution for the OSE smell using the metrics GDC, ADC, GBC and SDZ.

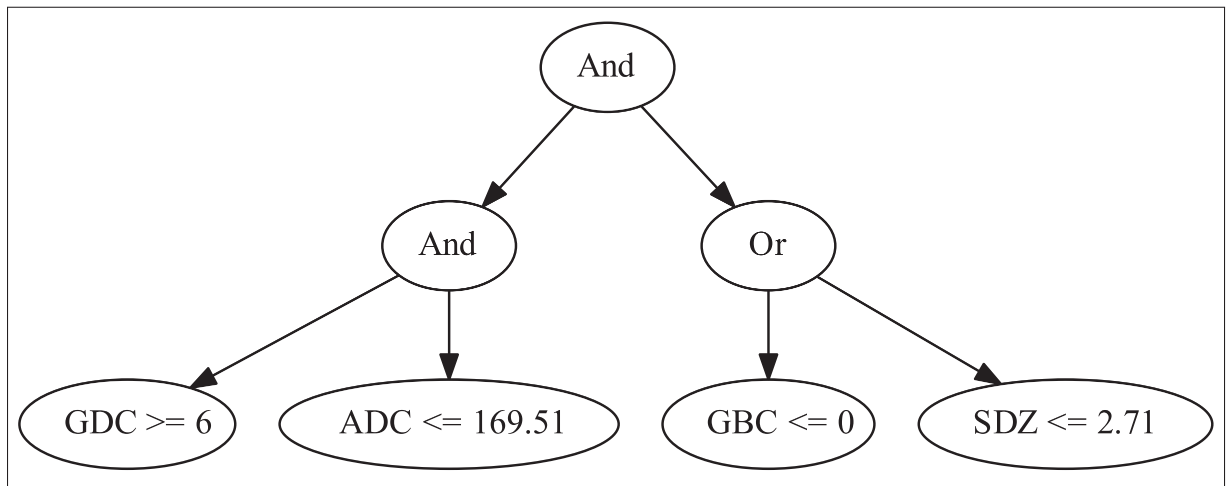


Figure 4.4 A simplified example of a solution for OSE smell.

(***ii***) **Generation of the initial population.** The initial population of solutions is generated randomly by assigning a variety of metrics and their thresholds to the set of different nodes of the tree. The size of a solution, *i.e.*, the tree's length, is randomly chosen between lower and upper bound values. These two bounds have been determined and called the problem of bloat control in GP, where the goal is to identify the tree size limits. Thus, we applied several trial and error experiments using the HyperVolume (HP) performance indicator Angeline (1994) to determine the upper bound after which, the sign remains invariant.

(***iii***) **Fitness function.** The fitness function evaluates how good is a candidate solution in detecting community smells. Thus, to evaluate the fitness of each solution, we use two objective functions, based on two well-known metrics Kessentini & Ouni (2017); Ouni *et al.* (2017); Harman & Clark (2004), to be optimized, *i.e.*, precision and recall. The precision objective

function aims at maximizing the detection of correct community smells over the list of detected ones. The recall objective function aims at maximizing the coverage of expected community smells from the base of examples over the actual list of detected smells. Precision and recall of a solution S are defined as follows.

$$Precision(S) = \frac{|\{\text{Detected smells}\} \cap \{\text{Expected smells}\}|}{|\{\text{Detected smells}\}|} \tag{4.1}$$

$$Recall(S) = \frac{|\{\text{Detected smells}\} \cap \{\text{Expected smells}\}|}{|\{\text{Expected smells}\}|} \tag{4.2}$$

(*iv*) Change operators. Crossover and mutation are used as change operators to evolve candidate solutions towards optimality.

Crossover. We adopt the “standard" random, single-point crossover. It selects two parent solutions at random, then picks a sub-tree on each one. Then, the crossover operator swaps the nodes and their relative subtrees from one parent to the other. Each child thus combines information from both parents. Figure 4.5 shows a simplified example of a crossover change operator.

Mutation: It can be applied either to a function node or a terminal node. This operator can modify one or many nodes. For a selected solution, the mutation operator first randomly selects a node in the tree. Then, if the selected node is a terminal (metric), it is replaced by another terminal (metric or another threshold value); if the selected node is a function (AND-OR-XOR operators), it is replaced by a new function (*e.g.*, AND becomes OR). If a tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree. Figure 4.6 shows a simplified example of a mutation change operator.

ECC majority voting. As shown in Figure 4.2, for each CC, MOGP will generate an optimal rule for each community smell type, *i.e.*, binary detection. Then, ECC allows finding the best CC that provides the best MLL from all the trained binary models. Each CC_i model is likely

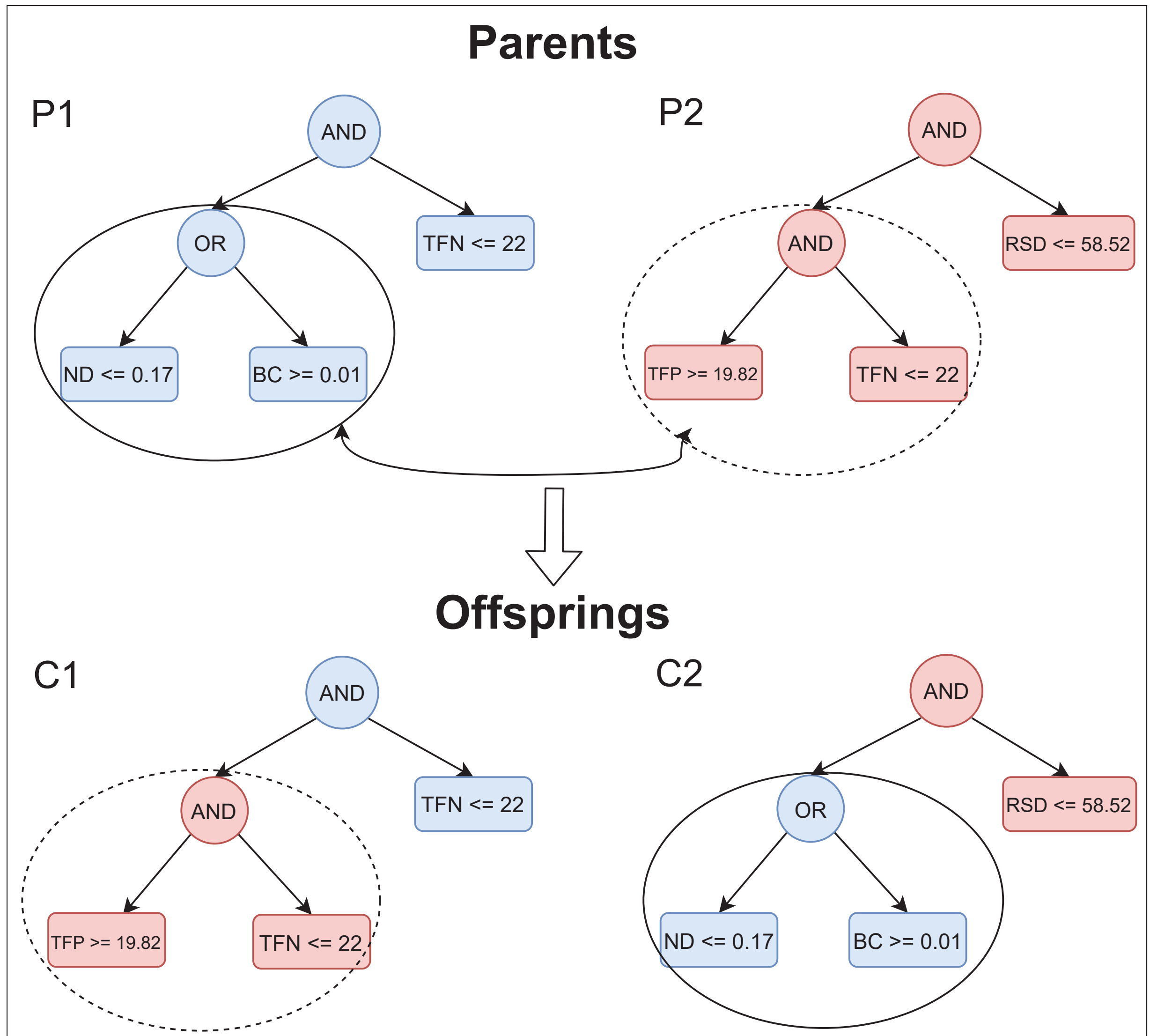


Figure 4.5 A simplified example of a crossover operator.

to be unique and able to achieve different multi-label classifications. These classifications are summed by label so that each label receives a number of votes. A threshold is used to select the most popular labels which form the final predicted multi-label set. This is a generic voting scheme and it is straightforward to apply an ensemble of any MLL transformation method Read *et al.* (2011).

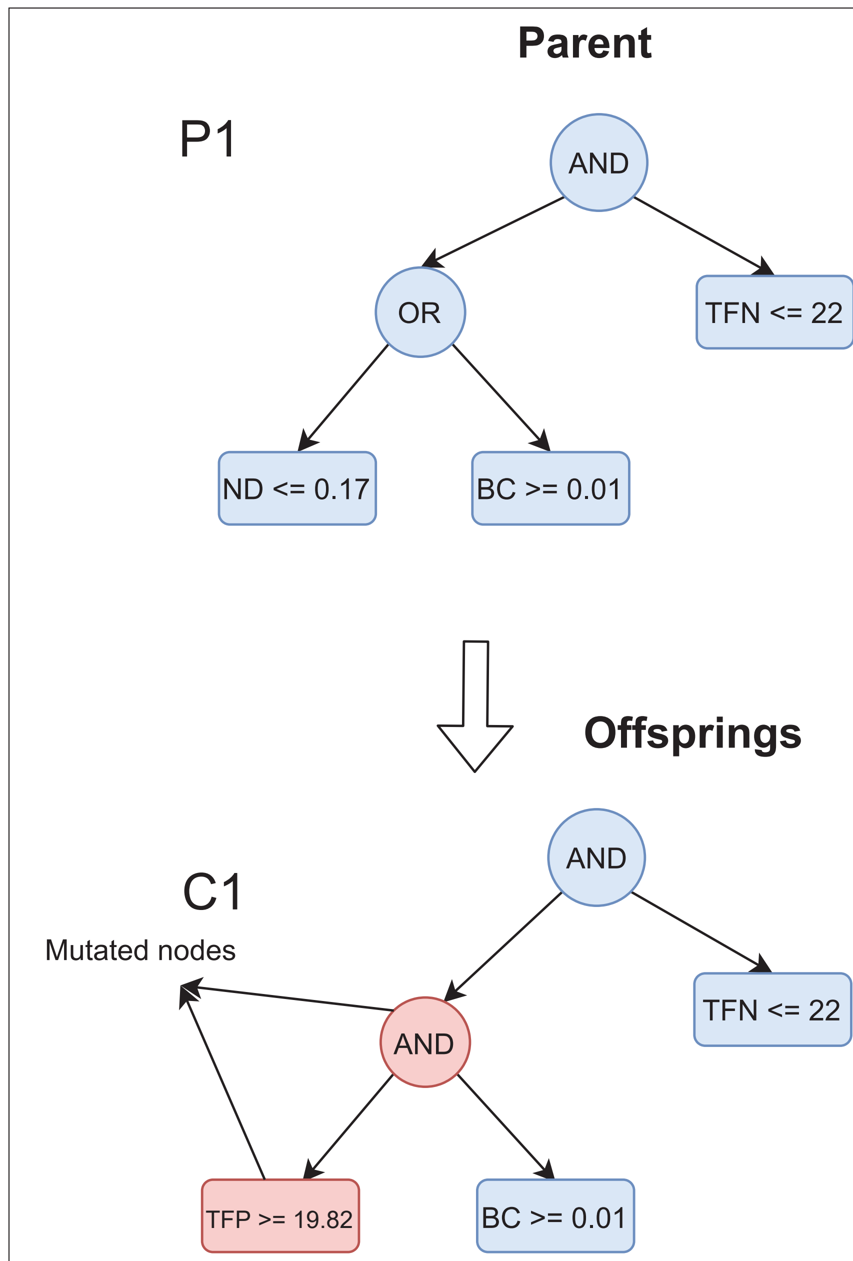


Figure 4.6 A simplified example of a mutation operator.

4.3.4 Step D : Detection Phase

After the GP-ECC model is constructed, in the training phase, it will be then used to detect a set of labels for a new project. It takes as input the set of features extracted from a given project

using the feature extraction module. As output, it returns the detection results for each individual label, *i.e.*, community smell type.

4.4 Empirical Evaluation

This section reports our empirical study to evaluate our extended approach eGP-ECC. We first define our research questions, then we describe our experiments setup and explain and discuss the obtained results.

4.4.1 Research Questions

We designed our empirical study to answer the four following research questions.

- **RQ1: (Performance)** How accurately can our GP-ECC approach detect community smells?
- **RQ2: (Sensitivity)** What types of community smells does our GP-ECC approach detect correctly?
- **RQ3: (Impact of the new sentiment and communication metrics)** Can the sentiment and communication-based metrics improve the performance of community smells detection?
- **RQ4: (Features influence)** What are the most influential features that can indicate the presence of community smells?

4.4.2 Analysis method

To evaluate our approach, we collected a set of community smells as discussed in Section 4.3.1. Table 4.3 summarizes the collected smells. Furthermore, as a sanity check, all smells were manually inspected and validated based on guidelines from the literature. Moreover, we conducted a survey with the original developers to help us to validate our identified symptoms. Furthermore, our dataset is available online for future extension and replication dat (2020).

We considered ten common types of community smells, *i.e.*, *organisational silo effect* (OSE), *black-cloud effect* (BCE), *prima-donnas effect* (PDE), *sharing villainy* (SV), *organisational skirmish* (OS), *solution defiance* (SD), *radio silence* (RS), *truck factor* (TF), *unhealthy interaction*

(UI), and *Unfriendly Communication* (UC), (cf. Section 4.2). In our experiments, we conducted a 10-fold cross-validation procedure to split our data into training data and evaluation data.

Table 4.3 Dataset statistics.

Data	Statistic
Number of projects	143
Number of projects having at least one smell	143
Total number of smells	696
Median number of smells per project	4
Median number of developers per project	42
Number of projects with <50 developers	82
Number of projects with 50 − 150 developers	46
Number of projects with >150 developers	15
Median number of commits per project	1035
Median number of issues per project	246.5
Median number of pull requests per project	371
Median number of days in each project	118.15
Number of projects written in Python	38
Number of projects written in Java	42
Number of projects written in Javascript	24
Number of projects written in C/C++/C#	26
Number of projects written in HTML/PHP/Ruby	13

To answer **RQ1**, we carry out a set of experiments to justify our eGP-ECC approach. We first compare the performance of our meta-algorithm ECC to two well-known meta-algorithms with proven success in MLL, *random k-labelset* (RAKEL) Tsoumakas & Katakis (2007) and *binary relevance* (BR) Tsoumakas & Katakis (2007). We next used GP, *decision tree* (J48) and *random forest* (RF) as their corresponding underlying classification algorithms. We also compared with the widely used MLL algorithm adaptation method, *K-Nearest Neighbors* (ML.KNN) Zhang & Zhou (2007). Thus, in total, we have 10 MLL algorithms to be compared. One fold is used for the test and 9 folds for the training.

To compare the performance of each method, we use common performance metrics, *i.e.*, precision, recall, and F-measure Read *et al.* (2011); Ouni *et al.* (2017); Kessentini & Ouni (2017); Xia *et al.* (2014). Let l a label in the label set L. For each instance i in the smells learning dataset, there are four outcomes, True Positive (TP_l) when i is detected as label l and it correctly belongs to l; False Positive (FP_l) when i is detected as label l and it actually does not

belong to l; False Negative (FN_l) when i is not detected as label l when it actually belongs to l; or True Negative (TN_l) when i is not detected as label l and it actually does not belong to l. Based on these possible outcomes, precision (P_l), recall (R_l) and F-measure (F_l) for label l are defined as follows:

$$P_l = \frac{TP_l}{TP_l + FP_l} \quad ; \quad R_l = \frac{TP_l}{TP_l + FN_l} \quad ; \quad F_l = \frac{2 \times P_l \times R_l}{P_l + R_l}$$

Then, the average precision, recall, and F-measure of the $|L|$ labels are calculated as follows:

$$Precision = \frac{1}{|L|}\sum_{l \in L} P_l \quad ; \quad Recall = \frac{1}{|L|}\sum_{l \in L} R_l \quad ; \quad F1 = \frac{1}{|L|}\sum_{l \in L} F_l$$

Statistical test methods. To compare the performance of each method, we perform Wilcoxon pairwise comparisons Cohen (2013) at 99% significance level (*i.e.*, $\alpha = 0.01$) to compare eGP-ECC with each of the 9 other methods. We also used the non-parametric effect Cliff's delta (d) Cliff (1993) to compute the effect size. The effect size d is interpreted as follows:

- Negligible if $|d| < 0.147$
- Small if $0.147 \leq |d| < 0.33$
- Medium if $0.33 \leq |d| < 0.474$
- High if $|d| \geq 0.474$

To answer **RQ2**, we investigated the efficiency of our eGP-ECC approach in detecting the different community smell types to find out whether there is a bias towards the detection of specific smell types. We use both precision and recall measures for each considered community smell in our study.

To answer **RQ3**, we aim at assessing the usefulness of our improved set of features that consists of (1) a set of 6 new developer's contribution metrics based on pull request and issue contributions

(cf. developer contribution metrics in Table 4.2), (2) a set of 5 new communication metrics based on pull requests and issues discussions (cf. communication metrics in Table 4.2), and (3) a set of 9 sentiment-based metrics (cf. sentiment analysis metrics in Table 4.2).

We hypothesize that extending our metrics suite allows to better capture the community smells properties along with extending the feature space for our eGP-ECC. We measure the smells detection performance, with and without the set of newly introduced metrics.

To answer **RQ4**, we aim at identifying the features that are the most important indicators of whether a project has a given community smell or not. For each smell type, we count the percentage of rules in which the feature appears across all obtained optimal rules by GP. The more a feature appears in the set of optimal trees, the more the feature is relevant to characterize that smell.

Algorithms parameters. For all the GP, RF and J48 algorithms, the maximum depth of the tree is set to 10. For GP, the population size is 200, number of iterations is 3,000, crossover and mutation rates are 0.9 and 0.1, respectively. For RF and J48, we used the default parameters of Weka. The number of neighbors of ML.KNN is set to 10. For ECC, we set the ensembles size $n = 20$. For RAKEL, we set $n = 20$, and the labels subset $k = 4$.

4.4.3 Results

This section reports and discusses our experimental results for each research question.

Results for RQ1 (Performance)

Table 4.4 reports the average precision, recall and F-measure scores for the 10 methods being compared. we observe that the meta-algorithm ECC outperforms the other 2 meta algorithms RAKEL and BR. Looking at the base learning methods (GP, J48 and RF), we used GP-ECC as the base for assessing statistical significance with each of the other methods. In particular, the GP-ECC method clearly achieved the highest F-measure with 0.93 as compared to the other methods with medium and large effect sizes, except with GP-RAKEL for which the F1

effect sizes were small. The same performance was achieved in terms of precision and recall, with 0.93 and 0.94, respectively. Moreover, we observe that GP-ECC achieves comparable performance as GP-RAKEL in terms of recall (with negligible effect size) which confirms the suitability of the GP formulation compared to decision tree and random forest algorithms. As for the binary relevance (BR) meta-classifier, we can also see overall superiority for GP-ECC compared to it in terms of precision, recall and F-measure with large effect size. BR is known to be a straightforward multi-label classification technique Tsoumakas & Katakis (2007), which decomposes the problem into a set of single-label multi-class problems. This simple method, however, totally neglects the potential dependencies among multiple labels, *i.e.*, community smell symptoms. In practice, the various community smells typically have interleaving characteristics and symptoms. For example, the Organizational Silo Effect (OSE) smell typically shares various socio-technical characteristics with the Solution Defiance (DF) smell which manifests in the form of independent subgroups in their development team social network due to the variance in their cultural and experience levels. Hence, the ECC and RAKEL meta-classifiers clearly outperform BR as they exploit the labels correlation along with an ensemble of classifiers to improve the detection performance. Moreover, among the 3 base learning algorithms, GP performs the best, followed by decision tree (J48) and random forest (RF).

Finally, the multi-label k nearest neighbor method (ML.KNN) turns out to be the wort method in terms of precision (0.84), recall (0.85) and F-measure (0.84). LkNN is not competitive with the other meta-classifiers, ECC, RAKEL and BR mainly due to the inadequacy of modeling label dependency. These findings advocate for the importance of leveraging multi-label learning with label dependency for the problem of community smells detection.

Results for RQ2 (Sensitivity) Figure 4 reports the sensitivity analysis for considered interleaving community smells. The figure shows the results for each specific community smell type, and overall we found that our approach is relatively stable across the different types of community smells. Overall, eGP-ECC achieved good performance and low variability in terms of both precision (ranging from 89% to 96%) and recall (ranging from 89% to 97%) across the 10 considered smell types. The highest precision and recall was obtained for the truck factor (TF)

Table 4.4 The achieved results by each of the meta-algorithms ECC, RAKEL and BR with their base learning algorithms GP, J48, and RF; and ML.KNN.

Algorithm	Precision		Recall		F1	
	score	p-value (d)*	score	p-value (d)*	score	p-value (d)*
GP-ECC	**0.87**	-	**0.91**	-	**0.89**	-
J48-ECC	0.84	<0.01 (M)	0.89	<0.01 (S)	0.86	<0.01 (M)
RF-ECC	0.84	<0.01 (M)	0.87	<0.01 (M)	0.85	<0.01 (M)
GP-RAKEL	0.85	<0.01 (S)	**0.91**	No. Stat. Sig.	0.88	<0.01 (S)
J48-RAKEL	0.83	<0.01 (L)	0.88	<0.01 (M)	0.85	<0.01 (L)
RF-RAKEL	0.84	<0.01 (M)	0.86	<0.01 (M)	0.85	<0.01 (M)
GP-BR	0.83	<0.01 (L)	0.85	<0.01 (M)	0.84	<0.01 (L)
J48-BR	0.81	<0.01 (L)	0.82	<0.01 (M)	0.81	<0.01 (L)
RF-BR	0.82	<0.01 (L)	0.82	<0.01 (L)	0.82	<0.01 (L)
ML.KNN	0.82	<0.01 (L)	0.84	<0.01 (L)	0.83	<0.01 (L)

* p-value(*d*) reports the statistical difference (p-value) and effect-size (*d*) between GP-ECC and the algorithm in the current row.
The effect-size (*d*) is N : Negligible − S : Small − M : Medium − L : Large

and the organizational silo effect (OSE), which heavily relies on the notion of developers social network and sub-groups. This higher performance is reasonable since the existing guidelines Tamburri *et al.* (2021); Ferreira *et al.* (2016); Tamburri *et al.* (2016, 2015a, 2019c); Raman *et al.* (2020); Murgia *et al.* (2018); Ortu *et al.* (2015b); Tourani *et al.* (2014) rely heavily on the notion of social network and sentiments polarity analysis. However, for smells such as sharing villainy (SV) and solution defiance (SD), the notion of social network is less important and this makes this type of smell harder to detect using such information.

To get a more qualitative sense, we present in Table 4.5 three examples from our experiments that cover various community smell types. Looking for instance at the `tensorflow/ranking` project, we observe that this project is affected by three community smells, namely the organizational silo effect (OSE), Black-cloud Effect (BCE), and Prima-donnas Effect (PDE). In particular, the project has been developed in a time window that starts on December 3, 2018 as a first commit until August 8, 2020 with a lifetime of 624 active days involving 20 developers when we analyzed it. Firstly, the key differentiating attributes for the OSE smell include 1) the low

degree centrality of the social network having a score of 0.25, which means that the immediate possibility that a developer can smoothly capture information through the project is low, 2) a low total number of sponsored developers (8 sponsored developers) which is typically correlated with low attractiveness and health of an Open-Source community as pointed out by prior research Tamburri *et al.* (2019); Fenton & Bieman (2019). Secondly, the BCE smell is characterized by 1) a low betweenness centrality oof 0.07, which is associated with additional occurrences of BCE smell as suggested by Tamburri et al. Tamburri *et al.* (2019), 2) a low social network density score of 0.38, which classifies the social network as a sparse graph indicating low communications between developers. 3) the subdivision of community members into 3 different sub-communities within the project which typically results into isolated communications among developers Tamburri *et al.* (2019). Thirdly, the identification pattern of the PDE smell is alsoo based on the existence of isolated sub-communities that have a lack of cooperation and communication in a project. The main community measures that characterize this smell include 1) a low closeness centrality having a low value of 0.13 which indicates the average number of steps that information has to take in order to reach every other node belonging to the social network, 2) a high number of time zones, with a value of 9, which increases the number of different levels of cultural and backgrounds in the development team.

Looking at the organizational structure of the `google/tangent` project (Table 4.5), we observe that it is affected by three community smells, the Organizational Skirmish (OS), Truck Factor (TF), and Unhealthy Interaction (UI). In particular, the project has been developed in a time window that starts on November 1, 2017 as a first commit until August 8, 2018 with a life time of 280 active days involving 15 developers when we analyzed it. The OS smell is measured by the modularity of the community based on the number of community sub-groups. This project has 3 communities with high standard deviation of commits per community (SCC) of 36.66 and a low average number of developers per community (ADC) of 5, and a high standard deviation of commits per time zone (SCZ) of 15.49 which indicates a misalignment between different expertise levels and communication channels between developers. On the other side, the TF smell manifests in the form of few developers having high density in the developers

social network. Indeed, we observe in this project that 5 developers have a coverage percentage of 45.19% which indicates that the project's information and knowledge are concentrated in few developers. Furthermore, a few comments in issues and pull requests channels lead to low developers participation in the project discussions and lead to UI smell, whereas average number of comments per PR (ANCPR) 1.27, and average number of comments per issue report (ANCI) 2.63.

We also observe that the `Microsoft/BotFramework-Composer` project experiences various instances of community smells including the Black-cloud Effect (BCE), Sharing Villainy (SV), Solution Defiance (SD), and Radio Silence (RS). In particular, the project has been developed in a time window that started from February 13, 2019 to July 10, 2020 with a life time of 512 active days involving 78 developers when we analyzed it. The BCE smell is characterized by 1) a low betweenness centrality of 0.01, which represents the degree to which developers (i.e., nodes in the social network) stand between each other, and 2) a low closeness centrality of 0.12 that indicates the low connectivity of the developers social network. Furthermore, the main side effect of the sharing villainy (SV) smell is the limited opportunity for developers (face to face) meetings to share knowledge and meaningful experiences which manifests in this project by a total number of 78 developers (NoD), and 13 time zones (TZ) making synchronous developers communication hard. Furthermore, the high number of time zones presents different levels of cultural and experience background which increases the probability of having the SD smell.

Table 4.5 Examples of projects and their related community smells.

Project Name	OSE	BCE	PDE	SV	OS	SD	RS	TF	UI	UC
Tensorflow/ranking	✔	✔	✔							
Google/tangent					✔			✔	✔	
Microsoft /BotFramework-Composer		✔		✔		✔	✔			✔

Regarding in sentiments analysis, figures 4.8, 4.9, 4.10, 4.11 present examples from our dataset to show negative sentiments in open-source projects.

Results for RQ3 (Impact of the new sentiment and communication metrics) To better understand the impact of the sentiments polarity analysis and the improved social network

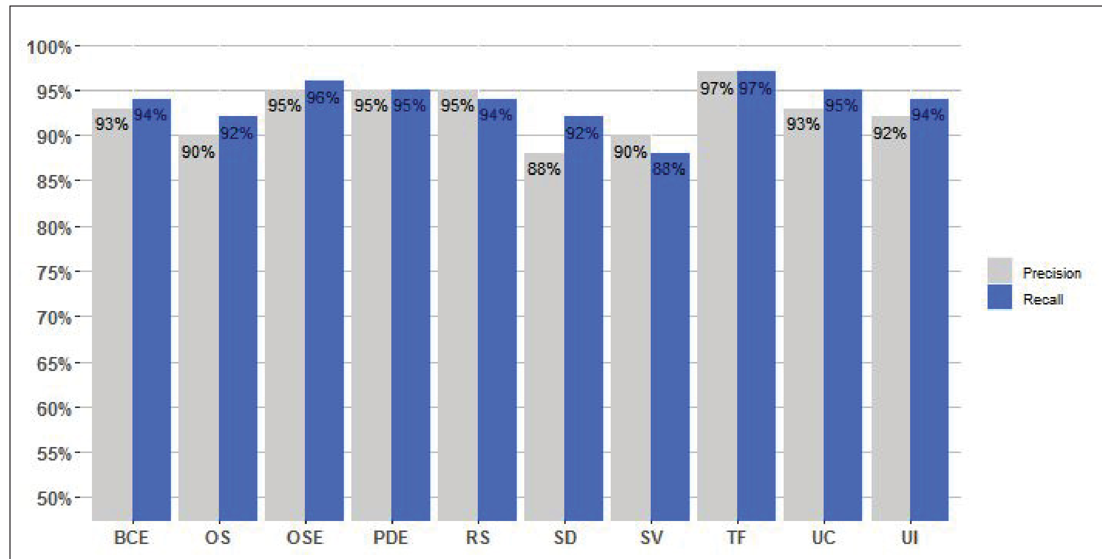


Figure 4.7 The achieved precision and recall scores by eGP-ECC for each community smell type.

incorporating both pull requests and issue tracking communications into our eGP-ECC model, we compare our approach with (eGP-ECC) and without (GP-ECC Almarimi *et al.* (2020a)) the metrics (cf. Table 4.2).

Figure 4.12 shows the results of both eGP-ECC and GP-ECC approaches for each community smell type, based on the detection performance in terms of precision and recall. Looking at the precision results from Figure 4.12-(a), we clearly observe that eGP-ECC achieves 92.9% which represents a significant improvement over GP-ECC achieving 79.7% for the ten considered community smell types. Similar finding are also observed in terms of recall from Figure 4.12-(b) where eGP-ECC and GP-ECC achieved 93.6%, and 81.6%, respectively. In particular, for the Black-cloud Effect (BCE) smell, the new communication and sentiment polarity metrics improved the detection precision from 83% (without the new metrics) to 94% (with the new metrics). We can also see a clear improvement in terms of the detection precision of eGP-ECC over GP-ECC for the Prima-donnas Effect (PDE), Solution Defiance (SD), Unhealthy Interaction (UI) and Unfriendly Communication (UC) smells. Indeed, in these particular community smells, the developer's social network and the quality of communications play an important role where

negative emotions or poor interactions may hinder developers from expressing themselves or may make them uncooperative when contributing in a project.

The obtained results provide more evidence on the importance of considering the analysis of various sources of communications among developers as well as the valuable knowledge embodied in the content of these communications to gain better understanding on the developers cooperative behaviors and quality of interactions.

To provide better understanding and get a more qualitative sense of the sentiments and emotions analysis, we report some illustrative examples from our dataset. Figure 4.8 shows developer comment that expresses sadness feeling from the `IntelLabs/nlp-architect` project, where the collaborator feels guilty towards the work. Furthermore, using the SentiStrength tool, the results show that this developer comment has negative sentiments (-1). We also observe in Figure 4.9 a comment that expresses a feeling of dissatisfaction towards work from the `android/android-test` project. The comment in Figure 4.10 is also extracted from the `android/android-test` project and expresses negative emotion (emotional tone 25.8). A negative tone below 50 indicated a more negative emotional tone based on LIWC tool analysis. Hence, the dominance of such negative emotions may hinder the productivity of developers and the quality of the project. Finally, the example in Figure 4.11 was selected from the Eclipse project, showing that brief comments with negative emotions may lead to poor communication practices among developers, and that is likely to make people leave a discussion.

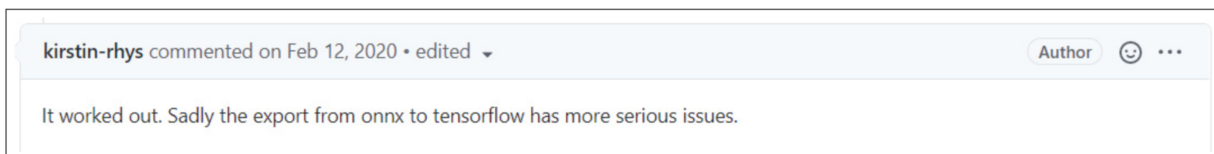


Figure 4.8 Sadness comment sentiments as an example.

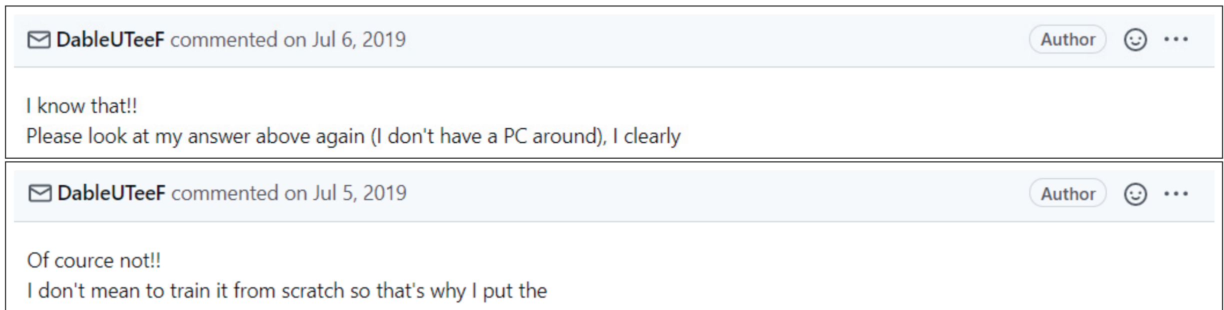


Figure 4.9 Anger comment sentiments as an example.

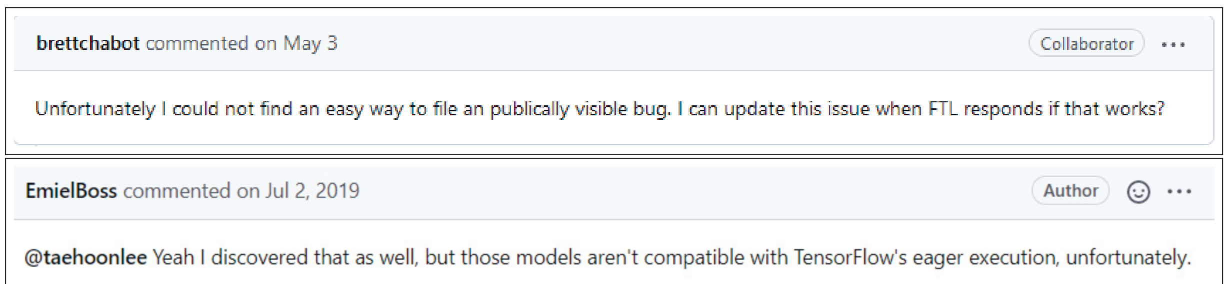
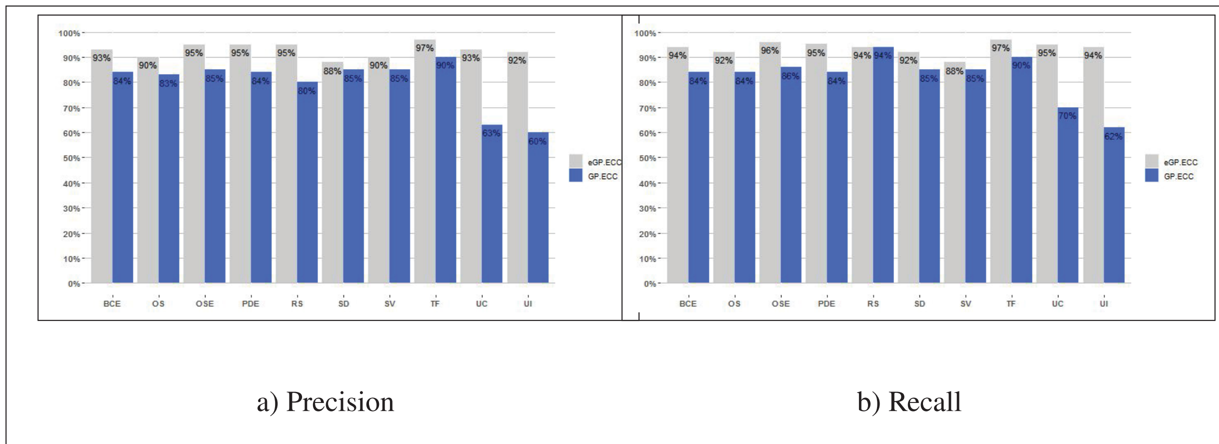


Figure 4.10 Negative emotion comment as an example.

Figure 4.11 Brief comment as an example.



a) Precision b) Recall

Figure 4.12 The achieved precision and recall results by eGP-ECC (with the new metrics) and GP-ECC (without the new metrics).

2

Results for RQ4 (Features influence)

To better understand what features are most selected by our eGP-ECC classifier to generate detection rules among all the generated rules, we count for each feature the percentage of optimal rules in which it appears. Our analysis is based on the assumption that the more a feature appears in the generated non-dominated smells detection rules, the more the feature is important for the detection of the corresponding smell Saidani, Ouni, Chouchen & Mkaouer (2020); Almarimi *et al.* (2020a); Saidani, Ouni & Mkaouer (2022).

Table 4.6 shows the statistics for each smell type with the top-20 features (cf. Table 4.2). We observe different community smell types have different influential features while sharing some other similar features.

For all the considered community smell types, we observe that the developers social network related metrics including the graph betweeness, closeness and degree centrality (GBC, GCC, and GDC) and the ratio the network density (ND) are the most influential in various community smell types such as the Organizational Silo Effect (OSE), Black-cloud Effect (BCE), the Organizational Skirmish (OS) and the Truck Factor (TF). Moreover, we observe that sentiment analysis related

metrics such as the ratio of issues with negative sentiments (RINC), the ratio of toxic comments in PR discussions (RTCPR), the ratio of anger words in PR discussions (RAWI) and the ratio of polite comments in PR discussions (RPCPR) are the most influential in various community smell types such as the Toxic Communication (TC), Solution Defiance (SD), Unhealthy Interaction (UI) and the Organizational Skirmish (OS). Furthermore, we also see from Table 4.6 that several direct communication and developer contribution metrics such the average number of authors per issue (ANA), the average number of comments per pull request (ANCPR), the standard deviation of developers per community and per time zone (SDC and SDZ) as well as the number of communities (NC) are influential in the vast majority of the community smell types.

These results indicate that different social network patterns, politeness, and anger related related aspects play a crucial role in the emergence of community smells. These findings suggest that more attention has to be paid to these particular socio-organizational characteristics within the software project community to avoid the presence of smells and their impact on the software project which may help software development communities to maintain healthy, productive and sustainable environment among the project participants.

Table 4.6 The most influential features for each community smell type.

Dimension	Metrics	OSE	BCE	PDE	SV	OS	SD	RS	TF	UI	TC
Community metrics	Average number of comments per pull request (ANCPR)	88	89	72	98	91	93	79	63	98	68
	Average number of comments per issue report (ANCI)	87	91	79	95	89	89	72	89	98	59
	Number of communities (NC)	79	62	72	43	52	62	95	72	89	64
Developer Contributions metrics	Average number of authors per issue report (ANAI)	92	93	81	80	81	79	82	55	88	81
	Average number of authors per pull request (ANAPR)	91	81	73	91	88	78	62	85	89	89
	Average number of comments per pull request (ANCPR)	88	89	72	98	91	93	79	63	98	68
	Percentage of sponsored developers (PSD)	55	21	34	45	32	62	65	81	73	81
	standard deviation of commits per developer (SDC)	91	88	89	88	91	66	82	93	79	69
Geographic Dispersion metrics	standard deviation of developers per time zone (SDZ)	85	53	71	82	62	88	72	63	92	91
	Time zones (TZ)	78	48	72	62	53	92	97	47	88	82
Social Network Analysis metrics	Graph degree centrality (GBC)	91	92	89	71	91	92	92	93	88	68
	Graph betweeness centrality (GDC)	91	91	93	90	91	83	96	97	73	36
	Graph closeness centrality (GCC)	90	88	89	91	93	91	91	95	80	82
	Network density (ND)	95	81	82	88	91	92	92	91	93	66
Sentiment Analysis metrics	Average communication comments Length (ACCL)	92	88	60	95	89	61	58	68	100	63
	Ratio of anger words in PR discussions (RAWI)	58	90	89	92	92	97	71	81	90	96
	Ratio of polite comments in PR discussions (RPCPR)	63	71	94	91	94	96	60	91	91	99
	Ratio of negative sentiments in PR comments (RNSPRC)	73	66	89	88	86	92	34	90	83	97
	Ratio of issues with negative sentiments (RINC)	63	63	91	81	87	96	52	54	89	97
	Ratio of toxic comments in PR discussions (RTCPR)	52	66	88	89	90	95	42	87	85	98
Truck Number metrics	Truck factor number (TFN)	22	15	21	41	22	42	39	100	29	31

4.5 Threats to validity

Threats to construct validity could be related to the performance measures. We basically used standard performance metrics such as precision, recall and F-measure that are widely accepted in MLL and software engineering. Read *et al.* (2011); Ouni *et al.* (2017); Kessentini & Ouni (2017); Xia *et al.* (2014); Ouni *et al.* (2013b). Therefore, We believe there is little threat to construct validity. Moreover, threats to construct validity could be linked to the generalizability of our experimental results. To address this threat, we have followed guidelines from the literature Tamburri *et al.* (2021, 2016, 2015a, 2013b) to construct an oracle of 143 open-source software systems hosted on GitHub. Moreover, the selection of these active projects was driven by two factors. First, we focused on communities having at least 10 contributors and 60 commits performed in their history to properly observe the community smells. Secondly, we aimed at studying systems that are active to mitigate threats due to outdated issues. However, other projects can be considered to further explore different contexts. Moreover we manually validated each community smell instance while excluding all smell instances for which there no agreement from the individual authors. We also attempted to mitigate this threat to validity by conducting a survey with the original developers to make sure whether the observed symptoms correctly match with the developers perception. While this validation makes us confident of the reliability of our smell instances in our dataset, there could still be errors that we did not notice. Moreover, we believe it is important to further test the validity of the studied community smells as an appropriate conceptual framework for OSS and extend the specific the study to collect more developer perceptions of their communities.

Related to our survey instrument, there is potential that the survey may influence the replies from the respondents. To address this issue, we made sure to ask for free-form responses and we publicly share our survey and all of our anonymized survey responses. On the other side, developer's aliases can also be a potential threat. This does not exclude possible errors when combining developer aliases could lead to erroneous combinations. Another potential threat could be related to the selection of classification techniques. Although we use the GP, J48

and RF techniques which are known to have high performance, there are other techniques. To mitigate this threat, we plan to compare with other MLL techniques.

Threats to internal validity relate to errors in our experiments. We have double checked our experiments and the datasets collected following the literature guidelines and a survey with developers Tamburri *et al.* (2021, 2016, 2015a, 2013b). Indeed, toxicity and sentiment analysis are hard to capture and may involve subjective judgment. To avoid this issue, we used common sentiment analysis tools such as Google's Perspective API AI, Stanford's Politeness detector tool, Danescu-Niculescu-Mizil *et al.* (2013b), LIWC lexicon Tausczik & Pennebaker (2010), and the SentiStrength tool Thelwall *et al.* (2012) that have been recently used in software engineering Raman *et al.* (2020); Lin *et al.* (2018); Guzman *et al.* (2014); Murgia *et al.* (2018); Ortu *et al.* (2015b); Ribeiro *et al.* (2016). We also used widely acknowledged techniques and tools to collect our metric suite based on . While these tools are reliable and widely used still there could be errors in estimating sentiments that we did not notice. Moreover, our study is limited to commits messages pull request, and issue discussions on GitHub. It does not include other forms of communication, such as forums, mailing lists, or face-to-face interactions. To mitigate this issue, we plan to consider other developers communication platforms such as Slack, Discord, etc., mailing lists and other issue tracking systems such JIRA.

Threats to external validity relate to the generalizability of our results. All of our findings were derived from open source projects hosted on GitHub. To minimize the threat to external validity, we chose open source projects from different domains. Furthermore, we chose the projects as a sample of the universe of OSS with different characteristics (i.e., commit size, community size, programming languages, availability of communication channels such Pull Requests and issues discussions).Since there could be a large number of valid projects, we must select a convenient sample to be studied, given the required manual inspection to identify existing smells. While we considered four different known organizations such as Microsoft projects (53 projects), Google (16 projects), Tensorflow (16 projects), and Eclipse (12 projects), we also considered other random projects from different organizations (46 projects). The majority of the projects selected in this study showed high levels of corporate engagement so our results may not generalize to

other OSS or commercial. However, we do not claim the generalizability of our results, and we consider the validity of our results limited to the contexts of the studied projects. Future replications of our analysis on other projects and organizations are needed.

4.6 Chapter Summary

We introduced in this paper eGP-ECC, an automated approach to detect community smells in software projects. We formulate the problem as a multi-label learning problem using the ECC meta-algorithm with an underlying GP model. Our eGP-ECC method aims at generating detection rules for each smell type. We use GP to translate regularities and symptoms that can be found in real-world community smell examples into detection rules. A detection rule is a combination of socio-technical attributes/symptoms with their appropriate threshold values to detect various types of community smells. We evaluated our approach on a set of 143 projects and 696 smell instances across 10 common types of community smells. Results show that our eGP-ECC approach can identify all the considered community smell types with an average F-measure of 93% and outperforms 9 state-of-the-art MLL techniques that rely on different meta-algorithms (ECC, BR and RAKEL) and different underlying learning algorithms (GP, J48, and RF); and a transformation method ML.KNN. Moreover, we conducted a deep analysis to investigate the symptoms, *i.e.*, features, that are the best indicators of community smells. We find that the standard deviation of the number of developers per time zone and per community, and the social network betweenness, closeness and density centrality and the ratio of issues with negative sentiments, the ratio of toxic comments, anger words and polite comments in PR and issue discussions are the most influential characteristics.

In future work, we plan to extend our approach with more open-source and industrial projects to provide ampler empirical evaluation. We plan also to extend our approach to provide software project managers with community change recommendations to avoid social debt in their projects. We also plan to assess the impact of community smells on different aspects of software quality and evolution.

CHAPTER 5

DISCUSSION AND LESSONS LEARNED

5.1 Discussion

Our research addresses key research contributions related to community smells detection using the csDetector approach. Our machine learning-based method employing C4.5 demonstrates exceptional performance, achieving an average accuracy of 96.9% and an AUC of 0.94 in detecting community smells. csDetector tool surpasses two recent state-of-the-art techniques, Code-Face4smell and Truck Factor, exhibiting a high detection accuracy ranging from 94% to 98% for eight considered community smells. Additionally, our approach identifies five highly influential metrics, including the Ratio of commits per time zone, Ratio of developers per time zone, Ratio of developers per community, social graph betweenness centrality, and social graph closeness centrality, serving as robust indicators of community smells. Moreover, the introduction of new metrics significantly enhances the detection performance for specific community smell types while maintaining consistent performance for others. Finally, the sensitivity analysis reveals the stability of our models, with a median sensitivity to influential data instances at a low 1.15%. Overall, our findings underscore the effectiveness and robustness of the csDetector approach in advancing community smells detection methodologies.

For further investigate the challenges associated with the detection of community smells, we conducted an empirical study . Specifically, we focused on the issue of interleaving organizational and social symptoms that serve as indicators of community smell instances within a software project. To address this, we propose an automated technique for detecting community smells in software projects. We formulate the problem as a multi-label learning (MLL) problem to handle the interleaving symptoms of existing community smells, generating multiple smell detection rules capable of detecting various types of community smells. We utilize the ensemble classifier chain (ECC) technique, which transforms the detection task of multiple smell types into several binary classification problems for each individual smell type. Our study demonstrates that the proposed technique achieves an average F-measure of 89% in detecting the eight considered

smell types. These findings contribute to the improvement and adaptation of the community smells detection problem, thereby assisting in ensuring the quality and success of software projects.

To enable the early detection of potential instances of poor community management during the software development life cycle, efficient and automated techniques are essential. In our study, we conduct an empirical study using a benchmark comprising 143 open-source projects from GitHub. We enhance an existing framework for detecting community smells by integrating multiple data sources, including social network analysis, sentiment analysis, pull requests, and issue discussions. Our framework demonstrates its potential benefits in a real-life scenario by soliciting feedback from software engineering practitioners regarding the community smells observed in their projects. Our findings indicate that the proposed approach successfully identifies the ten considered community smell types with an average F-measure of 93%.

5.2 Lessons Learned

Importance of Community Structures: Understanding the social and organizational dynamics within open-source communities is crucial for successful software development, emphasizing the fundamental role of social structures alongside technical aspects. The rapid evolution of software projects involving diverse stakeholders brings challenges in maintaining quality and functionality, highlighting the vital need for continuous attention to the socio-managerial structure of the project. Poor organizational and social practices manifest as community smells, causing social debt and potentially impacting projects negatively. Early identification and resolution of these issues are paramount. Our research showcases the feasibility of automated machine learning-based approaches, such as csDetector, for efficient community smell detection, offering personalized models adaptable to specific project contexts and improving detection accuracy. Moreover, our approach identified key community-related metrics, including social network graph analysis, developer features, truck factor features, and geographic dispersion features. Furthermore, our contribution to the research community by providing a publicly

accessible dataset fosters collaboration and advances research in community smells and social debt within software engineering.

Automated Detection Challenges: Community smells detection requires automated approaches but defining detection rules and handling overlapping symptoms stance challenges. In addressing the challenges of automated community smells detection, our research employs advanced techniques to navigate complexities. Recognizing the intricate nature of community smells, we formulate the problem as a Multi-Label Learning (MLL) issue, allowing simultaneous detection of multiple smell types. The innovative use of Ensemble Classifier Chain (ECC) technique enhances the classification process by leveraging previously identified labels, showcasing the importance of ensemble methods in tackling intricate detection tasks. Genetic Programming (GP) becomes instrumental in generating optimal detection rules from real-world instances, translating patterns from community smell examples into effective rules. Through a comprehensive empirical study involving 103 software projects, our research validates the GP-ECC approach, demonstrating an average F-measure of 89% and outperforming state-of-the-art Single- and Multi-Label Learning techniques.

Role of Communication Channels: The effective detection of community smells necessitates a meticulous analysis of diverse communication channels among developers, encompassing sentiment analysis, pull requests, and issue discussions. Our research goes beyond conventional metrics, integrating a detailed examination of developers' communication avenues. By constructing a social network grounded in discussions from issues and pull requests, coupled with sentiments and emotions analysis, our study delves deep into the social dimensions of open-source projects, offering profound insights into developer interactions and enriching the comprehension of community smells. This comprehensive communication analysis identifies specific socio-technical attributes and communication metrics as influential indicators of community smells. Moreover, our research underscores the significance of continuous adaptation in detection methods. By expanding the study to incorporate additional community smell types, integrating new communication channels, enriching the metrics suite, and updating the dataset, our approach ensures the relevancy and adaptability of detection techniques to evolving

project dynamics. A robust experimental setup, including the introduction of new community smells, expanded social network analysis, and incorporation of features from sentiments analysis, enhances the depth and accuracy of your experiments, highlighting the importance of meticulous experimental design and dataset preparation in software engineering research. Furthermore, our work demonstrates the value of a comprehensive literature review, encompassing various aspects of community smells, open-source projects' health, and sentiments analysis. This thorough review not only informs our research but also aids in understanding the existing knowledge landscape, identifying research gaps, and situating our contributions within the broader context of related studies.

CONCLUSION AND RECOMMENDATIONS

6.1 Conclusion and Findings

The primary objective of this thesis is exploring and detecting community smells in software development, as well as propose approaches to aid developers and managers for the early detection of potential community smells in a software project. In this chapter, we provide a summary of the thesis by outlining the main work and contributions made in each chapter. Furthermore, we discuss future research directions concerning community smells detection within the software engineering domain.

The three chapters of this thesis collectively contribute to the overarching goal of exploring and detecting community smells in software development, with a focus on aiding early detection in software projects. Chapter 3 initiates the exploration by conducting a qualitative analysis of 74 open-source projects on GitHub, unveiling various organizational and social symptoms indicative of potential community smells. This lays the foundation for Chapter 4, where an empirical study is conducted to delve deeper into the challenges associated with detecting community smells. The proposed automated technique, formulated as a multi-label learning (MLL) problem and utilizing ensemble classifier chain (ECC) technique, demonstrates substantial success in detecting different types of community smells. Chapter 5 builds upon these findings, emphasizing the importance of efficient and automated techniques for early detection. The empirical study in this chapter, incorporates multiple data sources and new common communication channels including pull requests, issues reports, and sentiment analysis to further refine the community smells detection framework. The enhanced framework, validated on a benchmark of 143 open-source projects, showcases significant improvement with an average F-measure of 93% in identifying ten community smell types. Collectively, these chapters offer a comprehensive progression from initial exploration and understanding of community smells to advanced automated detection techniques. The discussions and insights from each chapter feed into the overarching narrative

of advancing community smells detection within the software engineering domain. Looking forward, the research paves the way for future investigations and refinements in the field of community smells detection, contributing to the ongoing improvement of software project management.

6.2 Future Work

While this Ph.D. work has made significant contributions to understanding the problem of detecting community smells in software development, there are still numerous unexplored avenues for future research. In the following, we outline some of the main directions for future work.

Enhancing the detection of community smells in open source projects platforms:

In addition to our main findings in this thesis, we believe that the techniques proposed for detecting community smells from GitHub can be further extended in future research to enhance and facilitate the investigation of other phenomena on open-source project platforms. Replicating and adapting our work on other open-source platforms would allow for the generalization of our approaches and results, benefiting a broader range of developers. To support such future work, we have provided detailed technical implementations of our experiments and approaches within this thesis. Furthermore, we have published our data and scripts to facilitate future replications.

Replication in an industrial setting:

The results presented in this thesis explore and address the challenges associated with the detection of community smells. However, it is important to note that these results are based on the analysis of open-source projects exclusively. While we have made every effort to select representative and large open-source project platforms such as GitHub, and utilized appropriate data analysis techniques to mitigate threats to internal validity, we believe that software project

managers need to understand how these community smells impact the quality of their projects. Conducting future research that investigates and studies the impact of community smells in an industrial setting would allow us to generalize our results further.

Investigation the relation between community smells and a software quality:

The work of this thesis focused on understanding the problem of community smell detection in open-source projects. We believe that there are additional challenges that extend beyond maintaining software quality and functionality, impacting the socio-managerial structure of the project. Hence, several researchers and practitioners have highlighted the criticality of evolving organizational aspects to prevent decay and, consequently, software project failures. In future work, we plan to assess the impact of community smells on various aspects of software projects, including code smells and continuous integration (CI).

Investigation the interactions in software engineering:

In this thesis, we propose approaches and techniques to detect and identify interactions between developers in open-source projects. Future research should investigate and characterize interactions in various aspects of software engineering. As a software product encompasses different stages like design, development, deployment, and maintenance, and involves specific roles such as programmers, systems analysts, and project managers, it is essential to distinguish between these roles and investigate the affect-related metrics for each group separately. Doing so can lead to more precise findings. By analyzing their interactions, we can determine which types of interactions contribute to a more positive working environment.

BIBLIOGRAPHY

(2020). Replication Package. Retrieved from: https://github.com/GP-ECC/community-smells.

(2021). Replication Package, https://github.com/stilab-ets/CS.

A. Gopal, T. M. & S. Krishnan , M. (2002, April). The role of software processes and communication in offshore software development. *Communications of the ACM April 2002*, pp. 1106-1113. Retrieved from: https://doi.org/10.1145/505248.506008.

AI, C. What if technology could help improve conversations online?, https://www.perspectiveapi.com/. Retrieved from: https://www.perspectiveapi.com/.

Almarimi, N. (2019). Learning to detect community smells. Retrieved from: https://github.com/communitysmells/replicationPackage.

Almarimi, N. (2021). csDetector.

Almarimi, N., Ouni, A., Chouchen, M., Saidani, I. & Mkaouer, M. W. (2020a). On the detection of community smells using genetic programming-based ensemble classifier chain. *15th IEEE/ACM International Conference on Global Software Engineering (ICGSE)*, pp. 43–54.

Almarimi, N., Ouni, A. & Mkaouer, M. W. (2020b). Learning to detect community smells in open source software projects. *Knowledge-Based Systems*, 204, 106201.

Almarimi, N., Ouni, A., Chouchen, M. & Mkaouer, M. W. (2021). csDetector: an open source tool for community smells detection. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1560–1564.

Aman, H., Burhandenny, A. E., Amasaki, S., Yokogawa, T. & Kawahara, M. (2017). A Health Index of Open Source Projects Focusing on Pareto Distribution of Developer's Contribution. *8th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pp. 29–34.

Andrade, S. & Saraiva, F. (2017). Principled evaluation of strengths and weaknesses in FLOSS communities: A systematic mixed methods maturity model approach. *IFIP International Conference on Open Source Systems*, pp. 34–46.

Angeline, P. J. (1994). Genetic programming and emergent intelligence. *Advances in genetic programming*, 1, 75–98.

Antoine, J.-Y., Villaneau, J. & Lefeuvre, A. (2014). Weighted Krippendorff's alpha is a more reliable metrics for multi-coders ordinal annotations: experimental studies on emotion, opinion and coreference annotation.

Avelino, G., Passos, L., Hora, A. & Valente, M. T. (2016a). A novel approach for estimating truck factors. *IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–10.

Avelino, G., Passos, L., Hora, A. & Valente, M. T. (2016b). A novel approach for estimating truck factors. *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–10.

Baeza-Yates, R., Ribeiro-Neto, B. et al. (1999). *Modern information retrieval*. ACM press New York.

Baeza-Yates, R. A. & Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R. et al. (2001). Manifesto for agile software development.

Begel, A., Khoo, Y. P. & Zimmermann, T. (2010). Codebook: discovering and exploiting relationships in software repositories. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 1, 125–134.

Begel, A., Herbsleb, J. D. & Storey, M.-A. (2012). The future of collaborative software development. *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work Companion*, pp. 17–18.

Bettenburg, N. & Hassan, A. E. (2010). Studying the impact of social structures on software quality. *2010 IEEE 18th International Conference on Program Comprehension*, pp. 124–133.

Bindrees, M. A., Pooley, R. J., Ibrahim, I. S. & Bental, D. S. (2014). How public organisational structures influence software development processes. *Journal of Computer Science*, 10(12), 2593.

Bird, C., Gourley, A., Devanbu, P., Gertz, M. & Swaminathan, A. (2006). Mining email social networks. *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 137–143.

Bird, C., Nagappan, N., Devanbu, P., Gall, H. & Murphy, B. (2009a). Does distributed development affect software quality? an empirical case study of windows vista. *Communications of the ACM*, 52(8), 85–93.

Bird, C., Nagappan, N., Gall, H., Murphy, B. & Devanbu, P. (2009b). Putting it all together: Using socio-technical networks to predict failures. *2009 20th International Symposium on Software Reliability Engineering*, pp. 109–119.

Breiman, L. (2001a). Random Forests. *Machine Learning*, 45(1), 5–32.

Breiman, L. (2001b). Random forests. *Machine learning*, 45, 5–32.

Caballero-Espinosa, E., Carver, J. C. & Stowers, K. (2022). Community smells—The sources of social debt: A systematic literature review. *Information and Software Technology*, 107078.

Calefato, F., Lanubile, F., Maiorano, F. & Novielli, N. (2018). Sentiment polarity detection for software development. *Empirical Software Engineering*, 23(3), 1352–1382.

Canbek, G., Sagiroglu, S., Temizel, T. T. & Baykal, N. (2017). Binary classification performance measures/metrics: A comprehensive visualized roadmap to gain new insights. *2017 International Conference on Computer Science and Engineering (UBMK)*, pp. 821–826.

Capiluppi, A., Lago, P. & Morisio, M. (2003). Characteristics of open source projects. *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pp. 317–327.

Cataldo, M. & Nambiar, S. (2009). On the relationship between process maturity and geographic distribution: an empirical analysis of their impact on software quality. *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 101–110.

Cataldo, M. & Nambiar, S. (2012). The impact of geographic distribution and the nature of technical coupling on the quality of global software development projects. *Journal of software: Evolution and Process*, 24(2), 153–168.

Cataldo, M., Herbsleb, J. D. & Carley, K. M. (2008). Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pp. 2–11.

Cataldo, M., Mockus, A., Roberts, J. A. & Herbsleb, J. D. (2009). Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6), 864–878.

Catolino, G., Palomba, F. & Tamburri, D. A. (2019a). The Secret Life of Software Communities: What we Know and What we Don't Know. *BENEVOL*.

Catolino, G., Palomba, F., Tamburri, D. A., Serebrenik, A. & Ferrucci, F. (2019b). Gender diversity and community smells: insights from the trenches. *IEEE Software*, 37(1), 10–16.

Catolino, G., Palomba, F., Tamburri, D. A., Serebrenik, A. & Ferrucci, F. (2019c). Gender diversity and women in software teams: How do they affect community smells? *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, pp. 11–20.

Catolino, G., Palomba, F., Tamburri, D. A., Serebrenik, A. & Ferrucci, F. (2019d). Gender diversity and women in software teams: How do they affect community smells? *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, pp. 11–20.

Catolino, G., Palomba, F., Tamburri, D. A., Serebrenik, A. & Ferrucci, F. (2019e). Gender diversity and community smells: insights from the trenches. *IEEE Software*, 37(1), 10–16.

Catolino, G., Palomba, F., Tamburri, D. A., Serebrenik, A. & Ferrucci, F. (2020). Refactoring community smells in the wild: the practitioner's field manual. *Proceedings of the acm/ieee 42nd international conference on software engineering: Software engineering in society*, pp. 25–34.

Catolino, G., Palomba, F., Tamburri, D. A. & Serebrenik, A. (2021). Understanding community smells variability: A statistical approach. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, pp. 77–86.

Chawla, N. V., Bowyer, K. W., Hall, L. O. & Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16, 321–357.

Christley, S. & Madey, G. (2007). Analysis of activity in the open source software development community. *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, pp. 166b–166b.

Cliff, N. (1993). Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3), 494.

Coelho, J. & Valente, M. T. (2017). Why modern open source projects fail. *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*, pp. 186–196.

Cohen, J. (2013). *Statistical power analysis for the behavioral sciences*. Routledge.

Cosentino, V., Izquierdo, J. L. C. & Cabot, J. (2015). Assessing the bus factor of Git repositories. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 499-503.

Cosentino, V., Izquierdo, J. L. C. & Cabot, J. (2015). Assessing the bus factor of git repositories. *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 499–503.

Crowston, K. & Howison, J. (2006). Assessing the health of open source communities. *Computer*, 39(5), 89–91.

Cusick, J. & Prasad, A. (2006). A practical management and engineering approach to offshore collaboration. *IEEE software*, 23(5), 20–29.

Damian A. Tamburri,Simone Gatti, S. I. E. D. N. (2016). Re-Architecting Software Forges into Communities: An Experience Report. *JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS*, pp. 1-26.

Danescu-Niculescu-Mizil, C., Sudhof, M., Jurafsky, D., Leskovec, J. & Potts, C. (2013a). A computational approach to politeness with application to social factors. *arXiv preprint arXiv:1306.6078*.

Danescu-Niculescu-Mizil, C., Sudhof, M., Jurafsky, D., Leskovec, J. & Potts, C. (2013b). A computational approach to politeness with application to social factors. *arXiv preprint arXiv:1306.6078*.

de Carvalho, A. C. P. L. F. & Freitas, A. A. (2009). A Tutorial on Multi-label Classification Techniques. In Abraham, A., Hassanien, A.-E. & Snášel, V. (Eds.), *Foundations of Computational Intelligence Volume 5: Function Approximation and Classification* (pp. 177–195).

De Choudhury, M. & Counts, S. (2013). Understanding affect in the workplace via social media. *Proceedings of the 2013 conference on Computer supported cooperative work*, pp. 303–316.

De Falco, I., Della Cioppa, A. & Tarantino, E. (2002). Discovering interesting classification rules with genetic programming. *Applied Soft Computing*, 1(4), 257–269.

De Stefano, M., Pecorelli, F., Tamburri, D. A., Palomba, F. & De Lucia, A. (2020). Splicing Community Patterns and Smells: A Preliminary Study. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pp. 703–710.

Deb, K., Pratap, A., Agarwal, S. & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2), 182–197.

Deshpande, A. & Riehle, D. (2008). The total growth of open source. *Ifip international conference on open source systems*, pp. 197–209.

Dittrich, Y., Nørbjerg, J., Tell, P. & Bendix, L. (2018). Researching cooperation and communication in continuous software engineering. *2018 IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 87–90.

Efron, B. (1983). Estimating the Error Rate of a Prediction Rule: Improvement on Cross-Validation. *Journal of the American Statistical Association*, 78(382), 316–331.

El Asri, I., Kerzazi, N., Uddin, G., Khomh, F. & Idrissi, M. J. (2019). An empirical study of sentiments in code reviews. *Information and Software Technology*, 114, 37–54.

Elisseeff, A. & Weston, J. (2001). A Kernel Method for Multi-Labelled Classification. *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, (NIPS'01), 681–687.

Feitelson, D. G. (2012). Perpetual development: a model of the Linux kernel life cycle. *Journal of Systems and Software*, 85(4), 859–875.

Feng, Y. & Chen, Z. (2012). Multi-label software behavior learning. *34th International Conference on Software Engineering (ICSE)*, pp. 1305–1308.

Fenton, N. & Bieman, J. (2019). *Software metrics: a rigorous and practical approach*. CRC press.

Ferreira, M., Avelino, G., Valente, M. T. & Ferreira, K. A. (2016). A Comparative Study of Algorithms for Estimating Truck Factor. *Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pp. 91–100.

Fontana, F. A., Zanoni, M., Marino, A. & Mäntylä, M. V. (2013). Code smell detection: Towards a machine learning-based approach. *2013 IEEE international conference on software maintenance*, pp. 396–399.

Fowler, M. (2018). *Refactoring*. Addison-Wesley Professional.

Freitas, A. A. (1997). A Genetic Programming Framework for Two Data Mining Tasks: Classification and Generalized Rule Induction. In Koza, J. & Deb, K. (Eds.), *Proceedings of the Second Annual Conference Genetic Programming* (pp. 96–101). Morgan Kaufmann. Retrieved from: https://kar.kent.ac.uk/21483/.

Freitas, A. A. (2009). A review of evolutionary algorithms for data mining. In *Data Mining and Knowledge Discovery Handbook* (pp. 371–400). Springer.

Freitas, A. A. (2013). *Data mining and knowledge discovery with evolutionary algorithms*. Springer Science & Business Media.

Gachechiladze, D., Lanubile, F., Novielli, N. & Serebrenik, A. (2017). Anger and its direction in collaborative software development. *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*, pp. 11–14.

Gamalielsson, J. & Lundell, B. (2011). Open Source communities for long-term maintenance of digital assets: what is offered for ODF & OOXML. *Proceedings of SOS*, pp. 19–24.

Gamalielsson, J. & Lundell, B. (2014). Sustainability of Open Source software communities beyond a fork: How and why has the LibreOffice project evolved? *Journal of Systems and Software*, 89, 128–145.

Garcia, D., Zanetti, M. S. & Schweitzer, F. (2013). The role of emotions in contributors activity: A case study on the Gentoo community. *2013 International conference on cloud and green computing*, pp. 410–417.

Glover, F. W. & Kochenberger, G. A. (2006). *Handbook of metaheuristics*. Springer Science & Business Media.

Goggins, S., Lumbard, K. & Germonprez, M. (2021). Open source community health: Analytical metrics and their corresponding narratives. *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*, pp. 25–33.

Goncalves, E. C., Plastino, A. & Freitas, A. A. (2013). A genetic algorithm for optimizing the label ordering in multi-label classifier chains. *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pp. 469–476.

Gonzalez-Barahona, J. M., Sherwood, P., Robles, G. & Izquierdo, D. (2017). Technical lag in software compilations: Measuring how outdated a software deployment is. *IFIP International Conference on Open Source Systems*, pp. 182–192.

Goul, M., Marjanovic, O., Baxley, S. & Vizecky, K. (2012). Managing the enterprise business intelligence app store: Sentiment analysis supported requirements engineering. *2012 45th Hawaii international conference on system sciences*, pp. 4168–4177.

Greenhoe, D. J. (2016a). Properties of distance spaces with power triangle inequalities. *arXiv preprint arXiv:1610.07594*.

Greenhoe, D. J. (2016b). Properties of distance spaces with power triangle inequalities. *arXiv preprint arXiv:1610.07594*.

Guzman, E. & Bruegge, B. (2013a). Towards emotional awareness in software development teams. *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pp. 671–674.

Guzman, E. & Bruegge, B. (2013b). Towards emotional awareness in software development teams. *9th joint meeting on foundations of software engineering*, pp. 671–674.

Guzman, E., Azócar, D. & Li, Y. (2014). Sentiment analysis of commit comments in GitHub: an empirical study. *Proceedings of the 11th working conference on mining software repositories*, pp. 352–355.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. & Witten, I. H. (2009a). The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1), 10–18.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. & Witten, I. H. (2009b). The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1), 10–18.

Harman, M. (2007). The current state and future of search based software engineering. 342–357.

Harman, M. & Clark, J. (2004). Metrics are fitness functions too. *10th International Symposium on Software Metrics*, pp. 58–69.

Harman, M. & Jones, B. F. (2001). Search-based software engineering. *Information and software Technology*, 43(14), 833–839.

Harman, M., Mansouri, S. A. & Zhang, Y. (2012a). Search-Based Software Engineering: Trends, Techniques and Applications. *ACM Comput. Surv.*, 45(1).

Harman, M., Mansouri, S. A. & Zhang, Y. (2012b). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1), 1–61.

Hastie, T., Tibshirani, R., Friedman, J. H. & Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction*. Springer.

Hata, H., Todo, T., Onoue, S. & Matsumoto, K. (2015). Characteristics of sustainable oss projects: A theoretical and empirical study. *2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 15–21.

He, H. & Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9), 1263–1284.

Herbsleb, J. D. & Mockus, A. (2003). An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on software engineering*, 29(6), 481–494.

Hofstede, G. J., Jonker, C. M. & Verwaart, T. (2008). Modeling power distance in trade. *International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pp. 1–16.

Hofstede, G. J., Jonker, C. M. & Verwaart, T. (2009). Modeling power distance in trade. *Multi-Agent-Based Simulation IX: International Workshop, MABS 2008, Estoril, Portugal, May 12-13, 2008, Revised Selected Papers 9*, pp. 1–16.

Hong, Q., Kim, S., Cheung, S. C. & Bird, C. (2011). Understanding a developer social network and its evolution. *2011 27th IEEE international conference on software maintenance (ICSM)*, pp. 323–332.

Hunt, K. J., Shlomo, N. & Addington-Hall, J. (2013). Participant recruitment in sensitive surveys: a comparative trial of ‘opt in’versus ‘opt out’approaches. *BMC Medical Research Methodology*, 13(1), 3.

Islam, M. R. & Zibran, M. F. (2016). Towards understanding and exploiting developers’ emotional variations in software engineering. *IEEE 14th International Conference on Software Engineering Research, Management and Applications (SERA)*, pp. 185–192.

Jaakkola, H. (2012). Culture sensitive aspects in software engineering. In *Conceptual Modelling and Its Theoretical Foundations* (pp. 291–315). Springer.

Jansen, S. (2014). Measuring the health of open source software ecosystems: Beyond the scope of project health. *Information and Software Technology*, 56(11), 1508–1519.

Joachims, T. (1998). Text categorization with support vector machines: Learning with many relevant features. *European conference on machine learning*, pp. 137–142.

Joblin, M., Mauerer, W., Apel, S., Siegmund, J. & Riehle, D. (2015a). From Developer Networks to Verified Communities: A Fine-Grained Approach. *37th IEEE International Conference on Software Engineering (ICSE)*, 1, 563-573.

Joblin, M., Mauerer, W., Apel, S., Siegmund, J. & Riehle, D. (2015b). From developer networks to verified communities: A fine-grained approach. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 1, 563–573.

John R. Koza, M. (1992). Genetic Programming: On Programming Computers by means of Natural Selection and Genetics. *MIT Press, Cambridge, MA, 1992.*

Jurado, F. & Rodriguez, P. (2015). Sentiment Analysis in monitoring software development processes: An exploratory case study on GitHub's project issues. *Journal of Systems and Software*, 104, 82–89.

Ke, W. & Zhang, P. (2010). The effects of extrinsic motivations and satisfaction in open source software development. *Journal of the Association for Information Systems*, 11(12), 5.

Kessentini, M. & Ouni, A. (2017). Detecting Android Smells Using Multi-Objective Genetic Programming. *IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 122-132.

Kessentini, M., Werda, W., Langer, P. & Wimmer, M. (2013). Search-Based Model Merging. *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, (GECCO '13), 1453–1460. doi: 10.1145/2463372.2463553.

Kinnear, K. E., Langdon, W. B., Spector, L., Angeline, P. J. & O'Reilly, U.-M. (1994). *Advances in genetic programming*. MIT press.

Koza, J. (1992). On the programming of computersby means of natural selection. MIT Press: Cambridge, MA, USA.

Krippendorff, K. (2018). *Content analysis: An introduction to its methodology*. Sage publications.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1060–1076.

Liaw, A. & Wiener, M. (2001). Classification and Regression by RandomForest. *Forest*, 23.

Liaw, A., Wiener, M. et al. (2002). Classification and regression by randomForest. *R news*, 2(3), 18–22.

Likert, R. (1932). A technique for the measurement of attitudes. *Archives of psychology*.

Lin, B., Zampetti, F., Bavota, G., Di Penta, M., Lanza, M. & Oliveto, R. (2018). Sentiment analysis for software engineering: How far can we go? *Proceedings of the 40th international conference on software engineering*, pp. 94–104.

Lin, B., Cassee, N., Serebrenik, A., Bavota, G., Novielli, N. & Lanza, M. (2022). Opinion mining for software development: a systematic literature review. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3), 1–41.

Lin, F.-r. & Chen, C.-h. (2004). Developing and evaluating the social network analysis system for virtual teams in cyber communities. *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*, pp. 8–pp.

Link, G. J. & Germonprez, M. (2018). Assessing open source project health. *24th Americas Conference on Information Systems 2018: Digital Disruption, AMCIS 2018*.

Lopez-Fernandez, L., Robles, G., Gonzalez-Barahona, J. M. et al. (2004). Applying Social Network Analysis to the Information in CVS Repositories. *MSR*, 2004, 1st.

Mäntylä, M., Adams, B., Destefanis, G., Graziotin, D. & Ortu, M. (2016). Mining valence, arousal, and dominance: possibilities for detecting burnout and productivity? *13th International Conference on Mining Software Repositories*, pp. 247–258.

Marinescu, R. (2004, Sep.). Detection strategies: metrics-based rules for detecting design flaws. *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 350-359. doi: 10.1109/ICSM.2004.1357820.

McIlroy, S., Ali, N., Khalid, H. & Hassan, A. E. (2016). Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering*, 21(3), 1067–1106.

McIntosh, A., Hassan, S. & Hindle, A. (2018). What can Android mobile app developers do about the energy consumption of machine learning? *Empirical Software Engineering*.

Meneely, A. & Williams, L. (2011a). Socio-technical developer networks: should we trust our measurements? *Proceedings of the 33rd International Conference on Software Engineering*, pp. 281–290.

Meneely, A. & Williams, L. A. (2011b). Socio-technical developer networks: should we trust our measurements? *2011 33rd International Conference on Software Engineering (ICSE)*, 281-290.

Mockus, A. & Herbsleb, J. D. (2002). Expertise browser: a quantitative approach to identifying expertise. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 503–512.

Molnar, C. (2020). *Interpretable machine learning*. Lulu. com.

Murgia, A., Tourani, P., Adams, B. & Ortu, M. (2014). Do developers feel emotions? an exploratory analysis of emotions in software artifacts. *11th working conference on mining software repositories*, pp. 262–271.

Murgia, A., Ortu, M., Tourani, P., Adams, B. & Demeyer, S. (2018). An exploratory qualitative and quantitative analysis of emotions in issue report comments of open source systems. *Empirical Software Engineering*, 23(1), 521–564.

Nagappan, N., Murphy, B. & Basili, V. (2008a). The Influence of Organizational Structure on Software Quality: An Empirical Case Study. *Proceedings of the 30th International Conference on Software Engineering*, pp. 521–530.

Nagappan, N., Murphy, B. & Basili, V. (2008b). The influence of organizational structure on software quality: an empirical case study. *Proceedings of the 30th international conference on Software engineering*, pp. 521–530.

Naparat, D., Finnegan, P., Cahalane, M. et al. (2015). Healthy community and healthy commons:‘opensourcing’as a sustainable model of software production. *Australasian Journal of Information Systems*, 19.

Navarro, G. (2001). A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1), 31–88.

Nordio, M., Estler, H. C., Meyer, B., Tschannen, J., Ghezzi, C. & Di Nitto, E. (2011). How do distribution and time zones affect software development? a case study on communication. *2011 IEEE Sixth International Conference on Global Software Engineering*, pp. 176–184.

Onoue, S., Kula, R. G., Hata, H. & Matsumoto, K. (2017). The health and wealth of OSS projects: Evidence from community activities and product evolution. *arXiv preprint arXiv:1709.10324*.

Ortu, M., Adams, B., Destefanis, G., Tourani, P., Marchesi, M. & Tonelli, R. (2015a). Are bullies more productive? Empirical study of affectiveness vs. issue fixing time. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 303–313.

Ortu, M., Adams, B., Destefanis, G., Tourani, P., Marchesi, M. & Tonelli, R. (2015b). Are bullies more productive? Empirical study of affectiveness vs. issue fixing time. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 303–313.

Ouni, A., Kessentini, M., Inoue, K. & Cinnéide, M. (2017). Search-Based Web Service Antipatterns Detection. *IEEE Transactions on Services Computing*, 10(4), 603-617.

Ouni, A. (2020). Search based software engineering: challenges, opportunities and recent applications. *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pp. 1114–1146.

Ouni, A., Kessentini, M., Sahraoui, H. & Hamdi, M. S. (2012). Search-based refactoring: Towards semantics preservation. *28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 347–356.

Ouni, A., Kessentini, M. & Sahraoui, H. (2013a). Search-based refactoring using recorded code changes. *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 221–230.

Ouni, A., Kessentini, M., Sahraoui, H. & Boukadoum, M. (2013b). Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1), 47–79.

Ouni, A., Kessentini, M., Sahraoui, H. & Hamdi, M. S. (2013c). The use of development history in software refactoring using a multi-objective evolutionary algorithm. *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pp. 1461–1468.

Ouni, A., Gaikovina Kula, R., Kessentini, M. & Inoue, K. (2015). Web service antipatterns detection using genetic programming. *Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pp. 1351–1358.

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K. & Deb, K. (2016a). Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3), 1–53.

Ouni, A., Kula, R. G. & Inoue, K. (2016b). Search-based peer reviewers recommendation in modern code review. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 367–377.

Ouni, A., Kula, R. G., Kessentini, M., Ishio, T., German, D. M. & Inoue, K. (2017). Search-based software library recommendation using multi-objective optimization. *Information and Software Technology*, 83, 55–75.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D. & Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 268-278.

Palomba, F. & Tamburri, D. A. (2021). Predicting the emergence of community smells using socio-technical metrics: A machine-learning approach. *Journal of Systems and Software*, 171, 110847.

Palomba, F., Tamburri, D. A., Serebrenik, A., Zaidman, A., Fontana, F. A. & Oliveto, R. (2018a). How do community smells influence code smells? *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 240–241.

Palomba, F., Tamburri, D. A., Fontana, F. A., Oliveto, R., Zaidman, A. & Serebrenik, A. (2018b). Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE transactions on software engineering*, 47(1), 108–129.

Palomba, F., Gennaro, F., Pavone, M., Aiello, N., Aiello, G. & Cacciato, M. (2019). Analysis of PCB parasitic effects in a Vienna Rectifier for an EV battery charger by means of Electromagnetic Simulations. *2019 21st European Conference on Power Electronics and Applications (EPE'19 ECCE Europe)*, pp. 1–10.

Panichella, S., Di Sorbo, A., Guzman, E., Visaggio, C. A., Canfora, G. & Gall, H. C. (2015). How can i improve my app? classifying user reviews for software maintenance and evolution. *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pp. 281–290.

Paradis, C. & Kazman, R. (2021). Design choices in building an MSR tool: The case of Kaiaulu. *1st International Workshop on Mining Software Repositories for Software Architecture*.

Peck, M. S. (2002). *The road less traveled: A new psychology of love, traditional values, and spiritual growth*. Simon and Schuster.

Pinzger, M., Nagappan, N. & Murphy, B. (2008a). Can Developer-module Networks Predict Failures? *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 2–12.

Pinzger, M., Nagappan, N. & Murphy, B. (2008b). Can developer-module networks predict failures? *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 2–12.

Pletea, D., Vasilescu, B. & Serebrenik, A. (2014). Security and emotion: sentiment analysis of security discussions on github. *Proceedings of the 11th working conference on mining software repositories*, pp. 348–351.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J. & Wang, B. (2003). Automated support for classifying software failure reports. *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 465–475.

Ralph, P., Chiasson, M. & Kelley, H. (2016). Social theory for software engineering research. *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pp. 1–11.

Raman, N., Cao, M., Tsvetkov, Y., Kästner, C. & Vasilescu, B. (2020). Stress and Burnout in Open Source: Toward Finding, Understanding, and Mitigating Unhealthy Interactions. *International Conference on Software Engineering, New Ideas and Emerging Results (ICSE-NIER)*.

Rastogi, A. (2016). Do biases related to geographical location influence work-related decisions in GitHub? *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 665–667.

Read, J., Pfahringer, B., Holmes, G. & Frank, E. (2011). Classifier chains for multi-label classification. *Machine learning*, 85, 333–359.

Ribeiro, M. T., Singh, S. & Guestrin, C. (2016). " Why should i trust you?" Explaining the predictions of any classifier. *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1135–1144.

Riehle, D., Riemer, P., Kolassa, C. & Schmidt, M. (2014). Paid vs. volunteer work in open source. *2014 47th Hawaii International Conference on System Sciences*, pp. 3286–3295.

Robles, G. & Gonzalez-Barahona, J. M. (2005). Developer identification methods for integrated data from various sources. *ACM SIGSOFT Software Engineering Notes*, 30(4), 1–5.

Rousinopoulos, A.-I., Robles, G. & González-Barahona, J. M. (2014). Sentiment analysis of free/open source developers: preliminary findings from a case study. *Revista Eletrônica de Sistemas de Informação*, 13(2).

Saeki, M. (1995). Communication, collaboration and cooperation in software development-how should we support group work in software development? *Proceedings 1995 Asia Pacific Software Engineering Conference*, pp. 12–20.

Saidani, I., Ouni, A., Chouchen, M. & Mkaouer, M. W. (2020). Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128, 106392.

Saidani, I., Ouni, A. & Mkaouer, W. (2022). Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering*.

Scholbeck, C. A., Molnar, C., Heumann, C., Bischl, B. & Casalicchio, G. (2020). Sampling, intervention, prediction, aggregation: a generalized framework for model-agnostic interpretations. *Machine Learning and Knowledge Discovery in Databases: International Workshops of ECML PKDD 2019, Würzburg, Germany, September 16–20, 2019, Proceedings, Part I*, pp. 205–216.

Sen, R., Singh, S. S. & Borle, S. (2012). Open source software success: Measures and analysis. *Decision Support Systems*, 52(2), 364–372.

Sharp, H., Robinson, H. & Woodman, M. (2000a). Software engineering: community and culture. *IEEE Software*, 17(1), 40–47.

Sharp, H., Robinson, H. & Woodman, M. (2000b). Software engineering: community and culture. *IEEE Software*, 17(1), 40–47.

Sinha, V., Lazar, A. & Sharif, B. (2016). Analyzing developer sentiment in commit logs. *Proceedings of the 13th international conference on mining software repositories*, pp. 520–523.

Souza, R. & Silva, B. (2017). Sentiment analysis of travis ci builds. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 459–462.

Steinmacher, I., Conte, T., Gerosa, M. A. & Redmiles, D. (2015). Social barriers faced by newcomers placing their first contribution in open source software projects. *18th ACM conference on Computer supported cooperative work & social computing*, pp. 1379–1392.

Sugar, W. (2014). *Studies of ID practices: A review and synthesis of research on ID current practices*. Springer.

Tamburri, D. A., Kruchten, P., Lago, P. & van Vliet, H. (2013). What is social debt in software engineering? *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 93-96.

Tamburri, D. A. A., Palomba, F. & Kazman, R. (2019). Exploring Community Smells in Open-Source: An Automated Approach. *IEEE Transactions on Software Engineering*, 1-1.

Tamburri, D., Kazman, R. & Van den Heuvel, W.-J. (2019a). Splicing community and software architecture smells in agile teams: An industrial study.

Tamburri, D. A. & Palomba, F. (2021). Evolving software forges: An experience report from Apache Allura. *Journal of Software: Evolution and Process*, e2397.

Tamburri, D. A., Lago, P. & Van Vliet, H. (2012). Uncovering latent social communities in software development. *IEEE software*, 30(1), 29–36.

Tamburri, D. A., Kruchten, P., Lago, P. & van Vliet, H. (2013a). What is social debt in software engineering? *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 93-96. doi: 10.1109/CHASE.2013.6614739.

Tamburri, D. A., Kruchten, P., Lago, P. & van Vliet, H. (2013b). What is social debt in software engineering? *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 93–96.

Tamburri, D. A., Lago, P. & Vliet, H. v. (2013c). Organizational social structures for software engineering. *ACM Computing Surveys (CSUR)*, 46(1), 1–35.

Tamburri, D. A., Kruchten, P., Lago, P. & Vliet, H. v. (2015a). Social debt in software engineering: insights from industry. *Journal of Internet Services and Applications*, 6(1), 10.

Tamburri, D. A., Kruchten, P., Lago, P. & Vliet, H. v. (2015b). Social debt in software engineering: insights from industry. *Journal of Internet Services and Applications*, 6, 1–17.

Tamburri, D. A., Kazman, R. & Fahimi, H. (2016). The architect's role in community shepherding. *IEEE Software*, 33(6), 70–79.

Tamburri, D. A., Palomba, F., Serebrenik, A. & Zaidman, A. (2018). Discovering community patterns in open-source: a systematic approach and its evaluation. *Empirical Software Engineering*.

Tamburri, D. A., Palomba, F. & Kazman, R. (2019b). Exploring community smells in open-source: An automated approach. *IEEE Transactions on software Engineering*, 47(3), 630–652.

Tamburri, D. A., Palomba, F., Serebrenik, A. & Zaidman, A. (2019c). Discovering community patterns in open-source: a systematic approach and its evaluation. *Empirical Software Engineering*, 24, 1369–1417.

Tamburri, D. A., Palomba, F. & Kazman, R. (2020). Success and failure in software engineering: A followup systematic literature review. *IEEE Transactions on Engineering Management*, 68(2), 599–611.

Tamburri, D. A., Palomba, F. & Kazman, R. (2021). Exploring Community Smells in Open-Source: An Automated Approach. *IEEE Transactions on Software Engineering*, 47(3), 630-652. doi: 10.1109/TSE.2019.2901490.

Tantisuwankul, J., Nugroho, Y. S., Kula, R. G., Hata, H., Rungsawang, A., Leelaprute, P. & Matsumoto, K. (2019). A topological analysis of communication channels for knowledge sharing in contemporary GitHub projects. *Journal of Systems and Software*, 158, 110416.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E. & Matsumoto, K. (2017). An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering*, 43(1), 1-18.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E. & Matsumoto, K. (2016). An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1), 1–18.

Tausczik, Y. R. & Pennebaker, J. W. (2010). The psychological meaning of words: LIWC and computerized text analysis methods. *Journal of language and social psychology*, 29(1), 24–54.

Thelwall, M., Buckley, K., Paltoglou, G., Cai, D. & Kappas, A. (2010). Sentiment strength detection in short informal text. *Journal of the American society for information science and technology*, 61(12), 2544–2558.

Thelwall, M., Buckley, K. & Paltoglou, G. (2012). Sentiment strength detection for the social web. *Journal of the American Society for Information Science and Technology*, 63(1), 163–173.

Thongtanunam, P., Shang, W. & Hassan, A. E. (2018). Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones. *Empirical Software Engineering*.

Thongtanunam, P., Shang, W. & Hassan, A. E. (2019). Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones. *Empirical Software Engineering*, 24, 937–972.

tool, S. CodeFcae4Smell. Retrieved from: https://github.com/maelstromdat/CodeFace4Smells.

Torchiano, M., Ricca, F. & Marchetto, A. (2011). Is my project's truck factor low? theoretical and empirical considerations about the truck factor threshold. *International Workshop on Emerging Trends in Software Metrics*, pp. 12–18.

Tourani, P., Jiang, Y. & Adams, B. (2014). Monitoring sentiment in open source mailing lists: exploratory study on the apache ecosystem. *CASCON*, 14, 34–44.

Tsay, J., Dabbish, L. & Herbsleb, J. (2014). Influence of social and technical factors for evaluating contribution in GitHub. *Proceedings of the 36th international conference on Software engineering*, pp. 356–366.

Tsirakidis, P., Kobler, F. & Krcmar, H. (2009). Identification of success and failure factors of two agile software development teams in an open source organization. *2009 Fourth IEEE International Conference on Global Software Engineering*, pp. 295–296.

Tsoumakas, G. & Katakis, I. (2007). Multi-label classification: An overview. *International Journal of Data Warehousing and Mining*, 3(3), 1–13.

Tsoumakas, G., Katakis, I. & Vlahavas, I. (2010a). Mining Multi-label Data. In Maimon, O. & Rokach, L. (Eds.), *Data Mining and Knowledge Discovery Handbook* (pp. 667–685).

Tsoumakas, G., Katakis, I. & Vlahavas, I. (2010b). Random k-labelsets for multilabel classification. *IEEE Transactions on Knowledge and Data Engineering*, 23(7), 1079–1089.

Turhan, B. (2012). On the dataset shift problem in software engineering prediction models. *Empirical Software Engineering*, 17, 62–74.

Umer, Q., Liu, H. & Sultan, Y. (2019). Sentiment based approval prediction for enhancement reports. *Journal of Systems and Software*, 155, 57–69.

Voria, G., Pentangelo, V., Della Porta, A., Lambiase, S., Catolino, G., Palomba, F. & Ferrucci, F. (2022). Community Smell Detection and Refactoring in SLACK: The CADOCS Project. *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 469–473.

Wang, H., Kessentini, M., Grosky, W. & Meddeb, H. (2015). On the Use of Time Series and Search Based Software Engineering for Refactoring Recommendation. *Proceedings of the 7th International Conference on Management of Computational and Collective IntElligence in Digital EcoSystems*, (MEDES '15), 35–42. doi: 10.1145/2857218.2857224.

Werder, K. & Brinkkemper, S. (2018). MEME-Toward a Method for EMotions Extraction from GitHub. *IEEE/ACM 3rd International Workshop on Emotion Awareness in Software Engineering (SEmotion)*, pp. 20–24.

Wu, D., Rosen, D. W., Panchal, J. H. & Schaefer, D. (2016). Understanding communication and collaboration in social product development through social network analysis. *Journal of Computing and Information Science in Engineering*, 16(1).

Xia, T., Fu, W., Shu, R. & Menzies, T. (2020). Predicting project health for open source projects (using the DECART hyperparameter optimizer). *arXiv preprint arXiv:2006.07240*.

Xia, X., Lo, D., Wang, X. & Zhou, B. (2013). Tag recommendation in software information sites. *Working Conference on Mining Software Repositories (MSR)*, pp. 287–296.

Xia, X., Feng, Y., Lo, D., Chen, Z. & Wang, X. (2014). Towards more accurate multi-label software behavior learning. *Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 134–143.

Yamashita, K., McIntosh, S., Kamei, Y. & Ubayashi, N. (2014). Magnet or sticky? an oss project-by-project typology. *Proceedings of the 11th working conference on mining software repositories*, pp. 344–347.

Yamashita, K., Kamei, Y., McIntosh, S., Hassan, A. E. & Ubayashi, N. (2016). Magnet or sticky? Measuring project characteristics from the perspective of developer attraction and retention. *Journal of Information Processing*, 24(2), 339–348.

Zar, J. H. (1972a). Significance Testing of the Spearman Rank Correlation Coefficient. *Journal of the American Statistical Association*, 67(339), 578-580.

Zar, J. H. (1972b). Significance testing of the Spearman rank correlation coefficient. *Journal of the American Statistical Association*, 67(339), 578–580.

Zhang, M.-L. & Zhou, Z.-H. (2007). ML-KNN: A lazy learning approach to multi-label learning. *Pattern recognition*, 40(7), 2038–2048.

Zhang, Y. & Hou, D. (2013). Extracting problematic API features from forum discussions. *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 142–151.